THESIS FOR THE DEGREE OF LICENTIATE OF PHILOSOPHY

Securing concurrent programs with dynamic information-flow control

PABLO BUIRAS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY AND GÖTEBORG UNIVERSITY

Göteborg, Sweden 2014

Securing concurrent programs with dynamic information-flow control
PABLO BUIRAS

Technical Report 118L
ISSN 1652-876X
Department of Computer Science and Engineering
Research group: Language-based security

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY and GÖTEBORG UNIVERSITY
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2014

Abstract

The work presented in this thesis focusses on dealing with timing covert channels in dynamic information-flow control systems, particularly for the LIO library in Haskell.

Timing channels are dangerous in the presence of concurrency. Therefore, we start with the design, formalisation and implementation of a concurrent version of LIO which is secure against them. More specifically, we remove leaks due to non-terminating behaviour of programs (termination covert channel) and leaks produced by forcing certain interleavings of threads, as a result of affecting their timing behaviour (internal timing covert channel). The key insight is to decouple computations so that threads observing the timing or termination behaviour of other threads are required to be at the same confidentiality level. This work only deals with internal timing that can be exploited through language-level operations. We also mitigate leaks that result from the precise measurement of the timing of observable events (external timing covert channel), e.g. by using a stopwatch.

Timing channels can also be exploited through hardware-based shared resources, such as the processor cache. This thesis presents a cache-based attack on LIO that relies on timing perturbations to leak sensitive information through internal timing. To address this problem, we modify the Haskell runtime to support instruction-based scheduling, a scheduling strategy that is indifferent to such perturbations from underlying hardware components, such as the cache, TLB, and CPU buses. We show this scheduler is secure against cache-based internal timing attacks for applications using a single CPU. Additionally, we provide a purely language-based implementation of the instruction-based strategy for LIO, by means of a library. We leverage the notion of resumptions, a restricted form of continuations, to control the interleaving of threads, forcing each thread to yield after every LIO operation. Due to the flexibility of this approach, we are able to support parallel computation in the library, a novel feature in information-flow control tools.

Finally, we present a new manifestation of internal timing in Haskell, by exploiting lazy evaluation to encode sensitive information as timing perturbations. We illustrate our claim with a concrete attack on LIO that relies on memoisation of shared thunks to leak information. We also propose a countermeasure based on restricting the implicit sharing of values.

Acknowledgments

The journey towards this thesis would not have been possible without the help and support of many people.

Firstly, I would like to thank my advisor Alejandro Russo for all his encouragement, his guidance, and for teaching me how to be a researcher. Thank you for the stimulating discussions, for helping me develop my ideas to fruition, and for the awesome teamwork. Most of all, thanks for all the fun we have working together. I look forward to our future collaboration on the road towards my PhD.

I would also like to thank my colleagues at the department for providing such a friendly work environment. To the people from our corridor, for the daily lunch expeditions and the fikas. Special thanks go to Dave and Wolfgang, for so many interesting conversations and for their good advice and feedback, both in academic and personal matters. To my awesome office mates, who are also dear friends: Arnar, thank you for Mouseless Mondays, Accent Fridays, and the Icelandic word of the day; Willard, thanks for all the fun we have at the office, and for teaching me to appreciate bad films.

I also want to thank my friends in Sweden for making my life better in so many ways. Thank you for listening to me and giving me advice. Thanks for the pub evenings, the films we watch together, the comedy nights, the board game evenings, afterwork, birthday parties, the barbecues, the beer tastings, the Jazz evenings, and basically every time we hang out. Bart, Raúl, Arnar, Willard, Mauricio, Daniel, Luciano, Hiva, Michał, Nick, it's been great fun spending time with you.

Bart, dankjewel voor de gezelligheid en de leuke tijd. Moet je zien, ik probeer om deze dankbetuigingen in het Nederlands te schrijven – ik ben bijna volwassen hoor! Bedankt voor "Portal bij de thee", de kinderliedjes, Parskell, de spannende discussies, de games, de grapjes, de katers, en in het algemeen voor alles wat je met mij wilt delen. Ook dat je mijn werk proefleest. Bedankt voor de goede zorgen, voor je steun en voor het verdragen van mijn eigenaardigheden. Maar bovenal, bedankt voor je steengoede vriendschap man, kei-gaaf!

Gracias a mis amigos argentinos que están lejos, Juan Manuel, Germán, Leo, Gerardo, Zeta, Juan, Eze, Taihú, Santiago, Martín, Alfredo; porque a pesar de las distancias, seguimos en contacto y sé que puedo contar con ustedes. A los ñoños, los SELENitas y los Barrieros, siempre los tengo presentes.

A mi familia, gracias por todo el apoyo incondicional y por el afecto que me brindan. Nada de esto sería posible sin ustedes. A mi hermano Ignacio, por estar siempre ahí, por compartir tantas cosas conmigo y porque cuando nos encontramos, es como si el tiempo no hubiera pasado.

CONTENTS

CHAPTER

ONE

INTRODUCTION

There is no arguing that Computer Science is one of the driving forces behind innovation and development in the modern world. No other science has ever managed to transform the world as a whole in such a radical way as computing technology has during the last half of the 20th century. Our lives changed dramatically as we entered the so-called Information Age, and we started to become more and more reliant on computers for everything, including critical tasks in our society such as managing the social security system or the banking system. Information, and the way it disseminates, is a crucial part of this infrastructure.

Nowadays, personal information has become a valuable commodity. Many people own a smart phone, where they can install and use *apps*, and access social media websites. These apps are usually given access to potentially sensitive information such as contacts, text messages, and notes. Leaking sensitive information to third-parties can have serious consequences for the lives of the users, so it is necessary to develop mechanisms to secure this information and control its propagation. The most widespread approach is known as *access control*, where the user must give explicit permission to the application to access sensitive information or functionalty. Once this access has been granted, there is no way of knowing how the application will use this information, and where it is propagated. For example, an app with both read access to the phone's contacts and Internet access might send the contacs to a server on the Internet without explicit consent from the user.

Information flow control (IFC) [14] is an alternative to access control that tracks how information is disseminated in a given program, and ensures that it is used according to a given policy. This thesis focusses on *dynamic information flow control*, which involves enforcing security at runtime by checking all potentially insecure operations as they are performed by the program. When such an operation occurs, the program

execution is stopped in some way, for example by simply aborting the program or, in some cases, by throwing a runtime exception.

Lately, concurrency has become a necessity for practical applications. In the last decade, multi-core processors have become commonplace, so programmers expect to leverage this capability by writing multi-threaded programs. However, in the context of an information-flow control system, naively adding concurrency introduces a new possibility to leak information through *covert channels* [8], i.e. leaking information by exploiting system features not intended for communication.

This work is developed in the context of a specific kind of dynamic enforcement, based on a *floating-label* approach, which borrows ideas from the operating systems security research community [19], and brings them into the field of language-based security. The main example of such an enforcement is LIO, a Haskell library for dynamic information-flow control which allows programmers to write programs with security guarantees.

An information-flow-aware system is usually pictured as having information that concerns a number of agents or *principals*. An information-flow policy specifies how these principals are related to each other, specifically in terms of how information is allowed to flow among them. There are two sides to information flow control: confidentiality and integrity of data. In the most typical scenario, we are mainly interested in confidentiality, i.e. ensuring that secret information is not visible to unauthorised principals. Information-flow control aims to provide *end-to-end* security [15].

The main contributions of this thesis revolve around extending LIO with concurrency, while protecting against timing covert channels. In what follows, we provide a brief overview of IFC, timing covert channels and LIO.

1 Information flow control

Information flow control first arose from the need to track the propagation of information in military contexts. The classic scenario for information-flow control is a system which contains both secret and public information, and we want to ensure that the public outputs of the program cannot be influenced by secret information. One way of enforcing this is to think of a program as having endpoints (inputs and outputs) where information is consumed and produced. A *label* is attached to each of these endpoints, which indicates whether the information at that point is public or secret. Whenever the program attempts a write operation into a public output, the information-flow control system must check whether the information that is being written comes from (or, more generally, depends on) any secret input. If that is the case,

the program is in violation of the security policy, and therefore considered insecure.

1.1 Policies

A *policy* is formalised as a relation among the security levels in the system, which specifies how information is allowed to flow between different levels. Typically, it is defined as a *lattice* structure [4] which induces an ordering relation, usually written $\sqsubseteq$. In general, $l_1 \sqsubseteq l_2$ means that information from level l_1 is allowed to flow to level l_2. The canonical example is the two-point lattice with two levels, L and H, which respectively stand for *low* (public) and *high* (secret), and where the only allowed flows are $L \sqsubseteq L$, $H \sqsubseteq H$, and $L \sqsubseteq H$. In general, the elements of the security lattice are used by the enforcement mechanism as labels for the information flowing through the system. The lattice elements can also be interpreted as actors/components in the system rather than just levels of confidentiality, which allows to express policies in a mutual distrust scenario [10, 17]. Fig. 1 shows an example of such a security lattice. The arrows indicate allowed flows. This policy allows public information to flow to hospitals (Public $\sqsubseteq$ Hospitals) and insurance companies (Public $\sqsubseteq$ Insurance), but it does not allow medical records in the hospital to be disclosed to the insurance companies.

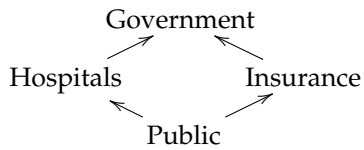

Fig. 1. Security lattice

1.2 Security property

In this setting, the kind of security properties we would like to guarantee are known as *noninterference* [6] properties. They can be informally stated in terms of a general malicious entity (the *adversary*), which has the ability to observe data below or at a given security level, but would like to observe confidential information not below this level. Then, we consider two independent runs of a program with inputs that are indistinguishable to the adversary, i.e. they only (potentially) differ in the parts that the adversary cannot see. We say that the program in question is *noninterfering* if the observable effects of these runs (outputs, return values, etc.) are also indistinguishable, as far as the adversary is concerned.

1.3 Enforcement mechanisms

In this thesis, we take a *dynamic* apprach to IFC. In a dynamic IFC system, the program is run alongside an *execution monitor*, a software component that is in charge of supervising the operations performed by the program (input/output in general) and checking that they comply with the security policy. The monitor will interrupt the execution of the program if a forbidden flow is detected. One example of a tool for dynamic IFC is JSFlow [7], an information-flow-aware JavaScript interpreter which runs as a browser plugin.

The alternative to the dynamic approach is *static information flow control* [18], which consists in statically analysing a program, just by examining its text, and classifying it as either secure or insecure depending on how information is propagated by it. There are several examples of static IFC systems, such as Jif [11] and Paragon [3]. They are both extensions to the Java language which allow programmers to express security policies, which are enforced using a type system.

2 Timing covert channels

Covert channels arise when information is leaked through mechanisms that were not originally designed for that purpose [8]. For example, the execution time of a program, the number of open files, or even the current volume of the speakers can be used by malicious programs to convey information to each other. In particular, timing channels affect the timing of programs to cause observable events to depend on secrets. Covert channels are, in some cases, easy to exploit if the adversary has access to the source code of the program, and especially so if the adversary is the one who writes it.

The main focus of this thesis is the *internal timing covert channel* [16]. This is a timing channel that exploits the interleaving of threads in a concurrent system to make the outcome of data races depend on sensitive information. The adversary can learn some of this information by observing these outcomes. In the rest of the thesis, the way in which the threads can encode bits of the secret into a data race usually depends on shared resources among the threads. Note that we do not assume the adversary to have the ability to precisely measure time when considering this channel.

The channel where the information is conveyed by the measurement of time (using a stopwatch) is known as *external timing covert channel*. Termination can be regarded as a special case, where the program takes an infinite amount of time to produce an output, and this fact can be used by the adversary to obtain sensitive information.

It has been shown that termination and timing channels are capable of leaking a considerable amount of information in a concurrent set-

ting [1]. For this reason, care must be taken when adding concurrency to an IFC system. In this work, we start with an existing dynamic IFC system, LIO, and show how to make it secure against certain classes of timing covert channels. The following section is a brief introduction to this IFC system.

3 LIO

LIO, which stands for *Labelled IO*, is a dynamic information flow control system for Haskell, a purely functional language with strong static typing [12]. Purity, or the absence of side-effects, means that pure code is only vulnerable to flows of information from parameters to return values. Additionally, the effectful part of an LIO program is written in an embedded language for describing computations, which allows direct control over all side-effects performed by the program. Unlike mainstream programming languages, where any effects are allowed anywhere by default, Haskell in principle *disallows* effects everywhere, except for special parts of the program where effects are explicitly marked by the type system. These blocks are written using *monads* [9], an abstract data type for specifying and combining effectful computations. As can be seen from earlier work [13], this makes Haskell a suitable language for information-flow analysis. LIO leverages this functionality to restrict the side-effects that the program can perform in order to enforce security.

LIO is embedded in Haskell as a *library*: programmers write their programs using the LIO interface, and their execution will also perform security checks to enforce a given policy. This library provides security guarantees in the form of noninterference properties, in the sense that every valid LIO program is noninterfering by construction (modulo covert channels). LIO is also parameterised in the security policy, which is specified as a lattice over a type of security labels.

LIO uses a *floating-label* approach to information flow control. An LIO computation has a *current label* attached to it, which is an upper bound on the sensitivity of the data in scope. When a computation with current label L_C observes an object A with label L_A, its current label must change (possibly rise) to the least upper bound or *join* of the two labels, written $L_C \sqcup L_A$. The current label effectively "floats above" the labels of all objects it observes. When performing a side-effect that will be visible to label L, LIO first checks that the current label flows to L ($L_C \sqsubseteq L$) before allowing the operation to take place.

Fig. 2 shows a basic example of how LIO works, assuming the security lattice from Fig. 1. The code is the definition of a malicious function $stealInfo$, which attempts to steal confidential medical information and send it to an insurance company. The function takes one argument, $medicalRecord$, which has the label Hospital, and is supposed to contain

the medical record of a person. This is an example of an LIO *labelled value*, which is simply a value protected by a security label. Labelled values must be unlabeled using the **unlabel** primitive, which returns the value itself and raises the current label of the LIO computation accordingly. In this example, we assume that the LIO computation has the label Public as its current label. After the first line has been executed, the current label of the computation would be Hospital. The medical record itself would be bound to m. In the second line, the program attempts to send[1] m to some insurance company, which has label Insurance, so the security check Hospital $\sqsubseteq$ Insurance is performed. Since the policy does not allow this flow, the program would be stopped at this point, and m would not reach $insuranceCompany$.

When writing LIO programs, one must be careful in the way that the program is structured and how the operations interact with the current label. It is a common mistake to unlabel too many values from several sources in the same context, inadvertently raising the current label so much that no useful outputs can be performed any more. This effect is known as *label creep*, and can be alleviated by a combination of mindful programming and a local scoping primitive called $toLabeled$. The changes to the current label in a $toLabeled$ block are undone after the block is executed, restoring the current label to what it was before running the block. In this sense, the floating-label approach seems to be a double-edged sword: on the one hand, it is perhaps a good idea to force programmers to structure their programs as a collection of blocks of code with different labels; on the other hand, the current label imposes a restrictive style that may constrain the programmer too much. Practical experience from the developers of GitStar [5], an information-flow-aware system built on top of LIO, seem to indicate that the model is appropriate for writing such systems and that the programmers did not feel overly constrained by it.

$$
\begin{aligned}
&stealInfo\ medicalRecord = \\
&\quad \mathbf{do}\ m \leftarrow \mathbf{unlabel}\ medicalRecord \\
&\qquad sendTo\ insuranceCompany\ m
\end{aligned}
$$

Fig. 2. Simple LIO code

4 Thesis overview

The contents of this thesis have been published as individual papers in the proceedings of peer-reviewed conferences and symposia. Each of

[1] The operation $sendTo$ does not exist as a primitive, but it is just meant as an example of an effectful operation that performs an output.

the four chapters that follow presents one of these papers. This section briefly outlines their contents and states the contributions of the author. Fig. 3 gives an overview of the four main chapters of the thesis. Chapters 2 and 5 deal with language-based covert channels, while Chapters 3 and 4 present two different ways of addressing hardware-based timing perturbations such as those caused by the processor cache.

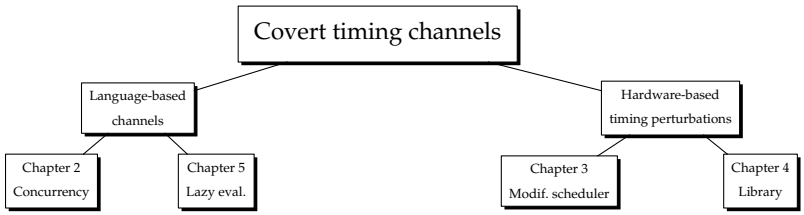

Fig. 3. Overview of the thesis

Chapter 2: Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems

As explained before, confidential information may be leaked through termination and timing channels. The termination covert channel has limited bandwidth for sequential programs [1], but it is a more dangerous source of information leakage in concurrent settings.

In this chapter, we present an information-flow control system that is secure against the termination and internal timing channels, i.e. situations in which the outcome of a race among several threads depends on confidential information. Intuitively, we leverage concurrency by placing such potentially sensitive actions in separate threads, each with its own floating label. Then, we require other threads to raise their current label accordingly before observing termination and timing of higher-confidentiality contexts. Additionally, we show how to mitigate external timing in this setting using ideas from Askarov et al [2]. The chapter introduces the concurrent version of LIO, which is, to the best of our knowledge, the first concurrent dynamic IFC system that deals with timing channels.

Statement of contributions The paper was co-authored with Deian Stefan, Alejandro Russo, Amit Levy, John C. Mitchell, and David Mazières. Pablo was mainly responsible for the contents of the Soundness section.

This chapter was published as a paper in the proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP) 2012.

Chapter 3: Eliminating cache-based timing attacks with instruction-based scheduling

In this chapter, we show that concurrent deterministic IFC systems that use time-based scheduling are vulnerable to a cache-based internal timing channel. We demonstrate this vulnerability with a concrete attack on LIO, which can be used to attack GitStar. The secret is encoded in the hardware cache, a resource that is implicitly shared among all threads and not modelled in the previous paper. As a result, the cache is not subject to LIO's usual IFC mechanisms, and the attack succeeds.

To eliminate this internal timing channel, we implement *instruction-based scheduling*, a new kind of scheduler that is indifferent to timing perturbations from underlying hardware components, such as the cache, TLB, and CPU buses. We show this scheduler is secure against cache-based internal timing attacks for applications using a single CPU.

Statement of contributions This paper was co-authored with Deian Stefan, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Pablo discovered the cache-based attack for LIO, contributed to the design of the cache-aware semantics and was responsible for the Semantics and Formal guarantees sections.

This chapter was published as a paper in the proceedings of the 18th European Symposium on Research in Computer Security (ESORICS) 2013.

Chapter 4: A library for removing cache-based attacks in concurrent information flow systems

In the previous chapter we present cache-based attacks in concurrent information flow systems, and provide a solution which involves modifying the scheduler in the Haskell runtime. In this chapter, we tackle the same problem from a purely language-based perspective, by providing a Haskell library that can be used as a replacement for concurrent LIO and which is resilient against cache-based attacks. We leverage *resumptions* – a tame form of continuations – to attain fine-grained control over the interleaving of thread computations at the library level. Specifically, we remove cache-based attacks by ensuring that every thread yields after executing an "instruction", i.e., an atomic action, in analogy with the behaviour of the instruction-based scheduler from the previous paper.

Statement of contributions This paper was co-authored with Amit Levy, Deian Stefan, Alejandro Russo and David Mazières. Pablo contributed to the design and implementation of the library, and was responsible for the formalisation and proofs.

This chapter was published as a paper in the proceedings of the 8th International Symposium on Trustworthy Global Computing (TGC) 2013.

Chapter 5: Lazy Programs Leak Secrets

Haskell's evaluation mechanism is *lazy*, which means that arguments to functions are not evaluated until they are needed in the body of the function. Crucially, when such an argument (also known as a *thunk*) is finally evaluated, its value gets cached and is reused in subsequent occurences of the same argument. In this chapter, we describe a novel exploit of lazy evaluation to reveal secrets in IFC systems through internal timing. We illustrate our claim with an attack on LIO. This attack is analogous to the cache-based attack, since thunks work like caches and can be shared by multiple threads by merely holding a pointer to them. We propose a countermeasure based on restricting the implicit sharing caused by lazy evaluation, but we do not implement these ideas, leaving them for future work.

Statement of contributions This paper was co-authored with Alejandro Russo. Pablo discovered the attack and contributed to the design of the proposed solution.

This chapter was published as a paper in the proceedings of the 8th Nordic Conference on Secure IT Systems (NordSec) 2013.

References

1. A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.
2. A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proc. of the 17th ACM CCS*. ACM, 2010.
3. N. Broberg, B. van Delft, and D. Sands. Paragon for practical programming with information-flow control. In C.-c. Shan, editor, *Programming Languages and Systems*, volume 8301 of *Lecture Notes in Computer Science*, pages 217–232. Springer International Publishing, 2013.
4. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
5. D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 47–60, Berkeley, CA, USA, 2012. USENIX Association.
6. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.
7. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. *Proc. 29th ACM Symposium on Applied Computing*, 2014.
8. B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, Oct. 1973.

9. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
10. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 129–142, 1997.
11. A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Computer Systems*, 9(4):410–442, October 2000.
12. S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.
13. A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell, 2008.
14. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.
15. J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in systems design. *ACM Trans. on Computer Systems*, 2(4):277–288, 1984.
16. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Prog. Languages*, Jan. 1998.
17. D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *Proc. of the NordSec 2011 Conference*, October 2011.
18. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, Jan. 1996.
19. N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

CHAPTER
TWO

ADDRESSING COVERT TERMINATION AND TIMING CHANNELS IN CONCURRENT INFORMATION FLOW SYSTEMS

Deian Stefan, Alejandro Russo, Pablo Buiras
Amit Levy, John C. Mitchell, David Mazières

Abstract. When termination of a program is observable by an adversary, confidential information may be leaked by terminating accordingly. While this termination covert channel has limited bandwidth for sequential programs, it is a more dangerous source of information leakage in concurrent settings. We address concurrent termination and timing channels by presenting an information-flow control system that mitigates and eliminates these channels while allowing termination and timing to depend on secret values. Intuitively, we leverage concurrency by placing such potentially sensitive actions in separate threads. While termination and timing of these threads may expose secret values, our system requires any thread observing these properties to raise its information-flow label accordingly, preventing leaks to lower-labeled contexts. We develop our approach in a Haskell library and demonstrate its applicability by implementing a web server that uses information-flow control to restrict untrusted web applications.

1 Introduction

Covert channels arise when programming language features are misused to leak information [28]. For example, when termination of a program is observable to an adversary, a program may intentionally or accidentally communicate a confidential bit by terminating according to the value of that bit. While this termination covert channel has limited bandwidth for sequential programs, it is a significant source of information leakage in concurrent settings. Similar issues arise with covert timing channels, which are potentially widespread because so many programs involve loops or recursive functions. These channels, based on either internal observation by portions of the system or external observation, are also effective in concurrent settings.

We present an information-flow system that mitigates and eliminates termination and timing channels in concurrent systems, while allowing timing and termination of loops and recursion to depend on secret values. Because the significance of these covert channels depends on concurrency, we fight fire with fire by leveraging concurrency to mitigate these channels: we place potentially nonterminating actions, or actions whose timing may depend on secret values, in separate threads. While termination and timing of these threads may expose secret values, our system requires any thread observing these properties to raise its information-flow label accordingly. We develop our approach in a Haskell library that uses the Haskell type system to prevent code from circumventing dynamic information-flow tracking. We demonstrate the applicability of this approach by implementing a web server that applies information-flow control to untrusted web applications. Although we do not address underlying hardware issues such as cache timing, our language-level methods can be combined with hardware-level mechanisms as needed to provide comprehensive defenses against covert channels.

Termination covert channel Askarov et al. [2] show that for sequential programs with outputs, the termination covert channel can only be exploited by exponentially complex brute-force: no attacker can reliably learn the secret in time polynomial in the size of the secret. Moreover, if secrets are uniformly distributed, the attacker's advantage (after observing a polynomial amount of output) is negligible in comparison with the size of the secret. Because of this relatively low risk, accepted sequential information-flow tools such as Jif [34], and FlowCaml [42], are only designed to address termination-insensitive noninterference. In a concurrent setting, however, the termination covert channel may be exploited more significantly [19]. We therefore focus on termination covert channels in concurrent programs and present an extension to our Haskell LIO library [46], which provides dynamic tracking of labeled values.

By providing labeled `forkLIO` and `waitLIO`, our extension removes the termination covert channel from sequential and concurrent programs while allowing loops whose termination conditions depend on secret information.

Internal timing channel Multi-threaded programs can leak information through an *internal timing covert channel* [50] when the observable timing behavior of a thread depends on secret data. This occurs when the time to produce a public event, such as placing public data on a public channel, depends on secret data, or, more generally, when a race to acquire a shared resource may be affected by secrets. We close this covert channel using the same approach as termination leaks: we decouple the execution of public events from computations that manipulate secret data. Using labeled `forkLIO` and `waitLIO`, computation depending on secret data proceeds in a new thread, and the number of instructions executed before producing public events does not depend on secrets. Therefore, a possible race to a shared public resource does not depend on the secret, eliminating internal timing leaks.

External timing channel External timing covert channels, which involve externally measuring the time used to complete operations that may depend on secret information, have been used in practice to leak information [7, 17] and break cryptography [18, 26, 51]. While several mechanisms exist to mitigate external timing channels [1, 5, 20], external timing channels are not addressed by conventional information-flow tools and in fact most of the previous techniques for language-based information-flow control appear to have limited application. Our contribution to external timing channels is to bring the mitigation techniques from the OS community into the language-based security setting. Generalizing previous work [3], Zhang et al. [53] propose a black-box mitigation technique that we adapt to a language-based security setting. In this approach, the source of observable events is wrapped by a timing mitigator that delays output events so that they contain only a bounded amount of information. We take advantage of the way Haskell makes it possible to identify when outputs are produced and implement the mitigator as part of the `LIO` library. Leveraging Haskell monad transformers [31], we show how to modularly extend `LIO`, or any other library performing side-effects in Haskell, to provide a suitable form of Zhang et al.'s mitigator.

In summary, the main contributions of this paper are:

- We present an information flow control (IFC) system that eliminates the termination and internal timing covert channels, while mitigating the external timing one. The system provides support for threads, light-weight synchronization primitives, and allows loops and branches to depend on sensitive (high) values. We believe this is the first

implementation of a language-based IFC system for concurrency that does not rely on cooperative-scheduling.

- We eliminate termination and internal-timing covert channels using concurrency, with potentially sensitive actions run in separate threads. This is implemented in a Haskell library that uses labeled concurrency primitives[1].
- We provide language-based support for resource-usage mitigation using monad transformers. We use this method to implement the black-box external timing mitigation approach of Zhang et al.; the method is also applicable to other covert channels, such as storage.
- We demonstrate the language implementation by building a simple server-side web application framework. In this framework, untrusted applications have access to a persistent key-value store. Moreover, requests to apps may be from malicious clients colluding with the application in order to learn sensitive information. We show several potential leaks through timing and termination and show how our library is used to address them.

Section 2.3 provides background on information flow, Haskell, and the Haskell LIO monad. We discuss the termination covert channel and its elimination in Section 3, the internal timing covert channel and its elimination in Section 4, and the external timing channel and its mitigation in Section 5. Formalization of the library is given in Section 6 and the security guarantees in Section 7. The implementation and experimental evaluation are presented in Section 8. Related work is described in Section 9. We conclude in Section 10.

2 Background

We build on a dynamic information flow control library in Haskell called LIO [46]. This section describes LIO and some of its relevant background. We first give an overview of IFC in abstract terms. We then give a brief overview of Haskell. Finally, we describe how LIO is implemented taking advantage of Haskell's static typing and pure functional nature.

2.1 Information flow control

IFC's goal is to track and control the propagation of information. In an IFC system, every observable bit has an associated *label*. Moreover, labels form a lattice [12] governed by a partial order $\sqsubseteq$ pronounced "can flow to." The value of a bit labeled L_{out} can depend on a bit labeled L_{in} only if $L_{\text{in}} \sqsubseteq L_{\text{out}}$.

[1] The library implementations discussed in this paper can be found at `http://www.scs.stanford.edu/~deian/concurrent_lio`

In a *floating-label* system, every execution context has a label that can rise to accommodate reading more sensitive data. For a process P labeled L_P to observe an object labeled L_O, P's label must rise to the least upper bound or *join* of the two labels, written $L_P \sqcup L_O$. P's label effectively "floats above" the labels of all objects it observes. Furthermore, systems frequently associate a *clearance* with each execution context that bounds its label.

Specific label formats depend on the application and are not the focus of this work. Instead, we will focus on a very simple two-point lattice with labels `Low` and `High`, where `Low` $\sqsubseteq$ `High` and `High` $\not\sqsubseteq$ `Low`. We, however, note that our implementation is polymorphic in the label type and any label format that implements a few basic relations (e.g., $\sqsubseteq$, join $\sqcup$, and meet $\sqcap$) can be used when building applications. The LIO library supports *privileges* which are used to implement decentralized information flow control as originally presented in [33]; though we do not discuss privileges in this paper, our implementation also provides privileged-versions of the combinators described in later sections.

2.2 Haskell

We choose the Haskell programming language because its abstractions allow IFC to be implemented in a library [29]. Building a library is far simpler than developing a programming language from scratch (or heavily modifying a compiler). Moreover, a library offers backwards compatibility with a large body of existing Haskell code.

From a security point of view, Haskell's most distinctive feature is a clear separation of pure computations from those with side-effects. Any computation with side-effects must have a type encapsulated by the monad `IO`. The main idea behind the LIO library is that untrusted actions must be specified with a new `LIO` monad instead of `IO`. Because the types are different, untrusted code cannot bind `IO` actions to `LIO` ones. The only `IO` actions that can be executed within `LIO` actions are the ones that have been wrapped in the `LIO` type using a private constructor only visible to trusted code. All such wrapped `IO` actions perform label checks to enforce IFC.

2.3 The `LIO` monad

In this section, we give an overview of LIO. LIO dynamically enforces IFC, but without the features described in this paper, provides only *termination-insensitive* IFC [2] for sequential programs. At a high level, LIO provides a monad called `LIO` (Labeled I/O) intended to be used in place of `IO`. The library furthermore contains a collection of `LIO` actions, many of them similar to `IO` actions from standard Haskell libraries, except that the `LIO` versions contain label checks that enforce IFC. For instance, LIO

provides file operations that look like those of the standard library, except that they confine the application to a dedicated portion of the file system where they store a label along with each file.

LIO is a floating-label system. The `LIO` monad keeps a *current label*, L_{cur}, that is effectively a ceiling over the labels of all data that the current computation may depend on. `LIO` also maintains a *current clearance*, C_{cur}, which specifies an upper bound on permissible values of L_{cur}.

LIO does not individually label definitions and bindings. Rather, all symbols in scope are identically labeled with L_{cur}. The only way to observe or modify differently labeled data is to execute actions that internally access privileged symbols. Such actions are responsible for appropriately validating and adjusting the current label.

As an example, the LIO file-reading function `readFile`, when executed on a file labeled L_F, first checks that $L_F \sqsubseteq C_{\text{cur}}$, throwing an exception if not. If the check succeeds, the function raises L_{cur} to $L_{\text{cur}} \sqcup L_F$ before returning the file content. The LIO file-writing function, `writeFile`, throws an exception if $L_{\text{cur}} \not\sqsubseteq L_F$.

As previously mentioned, allowing experimentation with different label formats, LIO actions are parameterized by the label type. For instance, simplifying slightly:

```
readFile :: (Label l) => FilePath -> LIO l String
```

To be more precise, it is really `(LIO l)` that is a replacement for the `IO` monad, where `l` can be any label type. The context `(Label l)=>` in `readFile`'s type signature restricts `l` to types that are instances of the `Label` typeclass, which abstracts the label specifics behind the basic methods $\sqsubseteq$, $\sqcup$, and $\sqcap$.

2.4 Labeled values

Since LIO protects all nameable values with L_{cur}, we need a way to manipulate differently-labeled data without monotonically increasing L_{cur}. For this purpose, LIO provides explicit references to labeled, immutable data through a polymorphic data type called `Labeled`. A locally accessible symbol (at L_{cur}) can name, say, a `Labeled l Int` (for some label type `l`), which contains an `Int` protected by a different label.

Several functions allow creating and using `Labeled` values:

- `label :: (Label l)=> l -> a -> LIO l (Labeled l a)`
  Given label $l : L_{\text{cur}} \sqsubseteq l \sqsubseteq C_{\text{cur}}$ and value v, action `label` l v returns a `Labeled` value guarding v with label l.

Listing 1 Exploiting the termination channel by brute-force

```
bruteForce:: String -> Int -> Labeled l Int -> LIO l ()
bruteForce msg n secret = forM_ [0..n] $ \i -> do
  toLabeled High $ do
    s <- unlabel secret
    if s == i then undefined else return ()
  outputLow (msg ++ show i)
```

- `unlabel :: (Label l)=> Labeled l a -> LIO l a`
  If `lv` is a `Labeled` value v with label l, `unlabel lv` raises L_{cur} to $L_{\text{cur}} \sqcup l$ (provided $L_{\text{cur}} \sqsubseteq C_{\text{cur}}$ still holds, otherwise it throws an exception) and returns v.
- `toLabeled :: (Label l) => l -> LIO l a -> LIO l (Labeled l a)`
  The dual of `unlabel`: given an action `m` that would raise L_{cur} to L'_{cur} where $L'_{\text{cur}} \sqsubseteq l \sqsubseteq C_{\text{cur}}$, `toLabeled` l `m` executes `m` without raising L_{cur}, and instead encapsulates `m`'s result in a `Labeled` value protected by label l.
- `labelOf :: (Label l)=> Labeled l a -> l`
  Returns the label of a `Labeled` value.

As an example, we show an `LIO` action that adds two `Labeled Int`s:

```
addLIO lA lB = do a <- unlabel lA
                  b <- unlabel lB
                  return (a + b)
```

If the inputs' labels are L_A and L_B, this action raises L_{cur} to $L_A \sqcup L_B \sqcup L_{\text{cur}}$ and returns the sum of the values.

We note that in an imperative language with labeled variables, dynamic labels can lead to implicit flows [13]. The canonical example is as follows:

```
public := 0;     // public has a Low label
if (secret)      // secret has a High label
  public := 1;   // public depends on secret
```

To avoid directly leaking the `secret` bit into `public`, one should track the label of the program counter and determine that execution of the assignment `public := 1` depends on `secret`, and raise `public`'s label when assigning `public := 1`. However, since the assignment executes conditionally depending on `secret`, now `public`'s *label* leaks the `secret` bit. LIO does not suffer from implicit flows. When branching on a secret, L_{cur} becomes `High` and therefore no public events are possible.

3 The termination covert channel

As mentioned in the introduction, information-flow control results and techniques for sequential settings do not naturally generalize to concur-

rent settings. In this section we highlight that the sequential LIO library allows leaks due to termination and show that a naive (but typical) extension that adds concurrency drastically amplifies this leak. We present a modification to the LIO library that eliminates the termination covert channel from both sequential and concurrent programs; our solution allows for flexible programming patterns, even writing loops whose termination condition depends on secret data.

Listing 1 shows an implementation of an attack (previously described by Askarov et al. in [2]) that leaks a secret in a brute-force way through the termination covert channel. Function `bruteForce` takes three arguments: a string message, the public maximum value that a non-negative secret `Int` can have, and a secret labeled `Int`. Given these arguments the function returns an `LIO` action that when executed returns unit `()`, but producing intermediate side-effects. Namely, `bruteForce` writes to a `Low` labeled channel using `outputLow` while L_{cur} is `Low`. We assume that `bruteForce` is executed with the initial L_{cur} as `Low`.

The attack consists of iterating (variable `i`) over the domain of the secret (`forM_ [0..n]`), producing a publicly-observable output if the guess, `i`, is not the value of the secret. When `i` is equal to the secret, the program diverges (`if s == i then undefined`). We use the constant `undefined` to denote any non-terminating computation. Observe that on every iteration L_{cur} is raised to the label of the secret within the `toLabeled` block. However, as described in Section 2.3, the current label outside the `toLabeled` block remains unaffected, and so the computation can continue producing publicly-observable outputs. The leak due to termination is obvious: when the attacker, observing the `Low` labeled output channel, no longer receives any data, the value of the secret can be inferred given the previous outputs. For instance, to leak a 16-bit bounded `secret`, we can execute `bruteForce "It is not: " 65536 secret`. Assuming the value of the secret is 4, executing the action produces the outputs "It is not: 0", "It is not: 1", "It is not: 2", "It is not: 3" before diverging. An observer that knows the implementation of `bruteForce` can directly infer that the value of the secret is 4. Observe that the code producing public outputs (`outputLow (msg ++ show i)`) does not inspect secret data at all, which makes it difficult to avoid termination leaks by simply tracking the flow of labeled data inside programs.

Suppose that we (naively) add support for concurrency to LIO using a hypothetical primitive `fork`, which simply spawns computations in new threads. Although we can preserve termination-insensitive noninterference (i.e., retain the property of no-explicit nor implicit-flows), we can extend the previous brute force attack to leak information in linear, as opposed to log, time in the length of the secret. In general, adding concurrency primitives in a straight-forward manner makes at-

Listing 2 A concurrent termination channel attack

```
concurrentAttack :: Int -> Labeled l Int -> LIO l ()
concurrentAttack k secret = forM_ [0..k] $ \i -> do
  iBit <- toLabeled High $ do
            s <- unlabel secret
            return (extractBit i s) )
  fork $ bruteForce (show k ++ "-bit:") 1 iBit
    where extractBit :: Int -> Int -> Int
          extractBit i n = (shiftR n i) .&. (bit 0)
```

tacks that leverage the termination covert channel very effective [19]. To illustrate this point, the attack of Listing 2 leaks the bit-contents of a secret value in linear time as follows. Given the bit-length `k` of a secret and the labeled `secret`, `concurrentAttack` returns an action which, when executed, extracts the bits of the secret (`extractBit i s`) and spawns a corresponding thread to recover them by executing the brute-force attack of Listing 1 (`bruteForce (show k ++ "-bit:") 1 iBit`). Hence, by collecting the public outputs generated by the different threads (having the form "0-bit:0", "1-bit:1", "2-bit:1", etc.), it is directly possible to recover the secret value.

3.1 Removing the termination covert channel in LIO

Since LIO is a floating-label system and at each point in the computation the evaluation context has a current label, a leak to a `Low` channel due to termination *cannot* occur after the current label is raised to `High`, unless the label-raise is within an enclosed `toLabeled` computation. Unless enclosed within a `toLabeled`, having L_{cur}=`High` implies that publicly-observable side-effects are no longer allowed. Hence, we can deduce that a piece of LIO code can exploit the termination covert channel only when using `toLabeled`. The key insight is that `toLabeled` is the single primitive in LIO that effectively allows a piece of code to temporarily raise its current label, perform a computation, and then continue with the starting current label. The attack in Listing 1 is a clear example that leverages this property of `toLabeled` to leak information.

Consider the necessary conditions for eliminating the termination channel present in Listing 1: the execution of the publicly-observable `outputLow` action must not depend on, or wait for, the secret computation executed within the `toLabeled` block. More generally, to close the termination covert channel, it is necessary to decouple the execution of computations enclosed by `toLabeled`. To achieve such decoupling, instead of using `toLabeled`, we provide an alternative primitive that executes computations that might raise the current label (as in `toLabeled`) in a newly-spawned thread. Moreover, to observe the result (or non-termination) of a spawned computation, the current label is firstly raised

to the label of the (possibly) returned result. In doing so, after observing a secret result (or non-termination) of a spawned computation, actions that produce publicly-observable side-effects can no longer be executed. In this manner, the termination channel is closed.

In Listing 1, the execution of `outputLow` is bound to the termination of the computation described by `toLabeled`. However, using our proposed approach of spawning a new thread when performing `toLabeled`, if the code following the `toLabeled` wishes to observe whether or not the `High` computation has terminated, it would first need to raise the current label to `High`. Thereafter, an `outputLow` action cannot be executed regardless of the result (or termination) of the `toLabeled` computation.

Concretely, we close the termination channel by removing the insecure function `toLabeled` from `LIO` and, instead, provide the following (termination sensitive) primitives.

```
forkLIO :: Label l => l -> LIO l a -> LIO l (Result l a)
waitLIO :: Label l => Result l a -> LIO l a
```

Intuitively, `forkLIO` can be considered as a concurrent version of `toLabeled`. `forkLIO l lio` spawns a new thread to perform the computation `lio`, whose current label may rise, and whose result is a value labeled with `l`. Rather than block, immediately after spawning a new thread, the primitive returns a value of type `Result l a`, which is simple a handler to access the labeled result produced by the spawned computation. Similar to `unlabel`, we provide `waitLIO`, which inspects values returned by spawned computations, i.e., values of type `Result l a`. The labeled wait, `waitLIO`, raises the current label to the label of its argument and then proceeds to inspect it.

In principle, rather than forking threads, it would be enough to prove that computations involving secrets terminate, e.g., by writing them in Coq or Agda. However, while this idea works in theory, it is still possible to crash an Agda or Coq program at runtime: for example, with a stack overflow. Generally, abnormal termination due to resource exhaustion exploits the termination channel and it could be hard to counter. In this light, forking threads is a manner to remove the termination channel by design. Although it might seem expensive, forking threads in Haskell is a light-weight operation[2].

We note that adding concurrency to `LIO` is a major modification which introduces security implications beyond that of handling the termination channel. In the following section, we describe the *internal timing covert channel*, a channel present in programming languages that have support for concurrency and shared-resources.

[2] http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Concurrent.html

Listing 3 Internal timing leak

```
sthread :: String -> Int -> Labeled l Bool -> LIO l ()
sthread msg n secret = do toLabeled High
                            ( do s <- unlabel secret
                                 if s then sleep n
                                 else return () )
                          outputLow msg

pthread :: String -> Int -> LIO l ()
pthread msg n = do sleep n
                   outputLow msg

attack :: Labeled l Bool -> LIO l ()
attack secret = do fork (sthread "True" 5000 secret)
                   fork (pthread "False" 1000)
```

4 The Internal timing covert channel

In a concurrent setting, the possibility that threads have to share resources opens up new information channels. Specifically, multi-threaded programs can leak information through the *internal timing covert channel* [50]. The source of the leaks comes from the ability of threads to affect their timing behavior based on secret data and thus affect, via the scheduler, the order of public events.

To illustrate internal timing attacks, we consider the LIO library from Section 2.3 with the added hypothetical primitive `fork` used to spawn a new thread. Listing 3 illustrates an internal timing attack. It consists of two threads: `sthread` and `pthread`. Command `sleep n` puts a thread to sleep for `n` milliseconds. Thread `sthread` takes a string to output in a public channel (`outputLow msg`) and the number of milliseconds to sleep (`sleep n`) if the secret boolean taken as argument (`secret`) is true. Thread `pthread`, on the other hand, does not take any secret but it writes a message (`msg`) to the same output channel as `sthread` after sleeping some milliseconds. Observe that both threads share a resource (i.e., the output channel) and that the timing behavior of `sthread` depends on the secret boolean.

With this example, `sthread` should take more time to execute the `outputLow` action than `pthread` if and only if the secret boolean is true. In isolation, both threads are secure, i.e., they satisfy non-interference. In fact, when considering them in isolation, both threads always produce the public output given by the argument `msg`. However, by running them concurrently, it is possible to leak information about `secret`. Function `attack` spawns two threads that execute `sthread` and `pthread` concurrently. Under many reasonable schedulers, if `secret` is true, it is more likely that the instruction `outputLow "False"` is executed first. On

the other hand, if `secret` is false, it is more likely that `outputLow "True"` is executed first. An attacker can then observe the value of `secret` by just observing the second produced output.

Unlike other timing channel attacks, internal timing attacks do not require an attacker to measure the actual execution time to deduce secret information. The interleaving of threads is simply responsible for producing leaks! Although the example in Listing 3 shows how to leak one bit, it is easy to place the attack in a loop that leaks bit by bit a whole secret value in linear time. Tsai et al. [48] show how effective the attack is even without having much information about the run-time system (e.g., the scheduler). The authors implemented the magnified version of the attack in Listing 3 and showed how to leak a credit card number.

4.1 Removing the internal timing channel

As indicated by the code in Listing 3, the internal timing covert channel can be exploited when the time to produce public events (e.g., sending some data in a public channel) depends on secrets. In other words, internal timing arises when there is a race to acquire a shared resource that may be affected by secret data. In order to close this channel, we apply the same technique as for dealing with termination leaks: we decouple the execution of public events from computations that manipulate secret data. By using `forkLIO` and `waitLIO`, computations dealing with secrets are spawned in a new thread. In that manner, any possible race to a shared public resource does not depend on the secret anymore and thus internal timing leaks are no longer possible.

4.2 Synchronization primitives in concurrent LIO

In the presence of concurrency, synchronization is vital. This section introduces an IFC-aware version of `MVar`s, which are well-established synchronization Haskell primitives [24]. As with `MVar`s, `LMVar`s can be used in different manners: as synchronized mutable variables, as channels of depth one, or as building blocks for more complex communication and synchronization primitives.

A value of type `LMVar l a` is mutable location that is either empty or contains a value of type `a` labeled with `l`. `LMVar`s are associated with the following operations:

```
newEmptyLMVar :: (Label l) => l -> LIO l (LMVar l a)
putLMVar      :: (Label l) => LMVar l a -> a -> LIO l ()
takeLMVar     :: (Label l) => LMVar l a -> LIO l a
```

Function `newEmptyLMVar` takes a label `l` and creates an empty `LMVar l a` for any desired type `a`. The creation succeeds only if the label `l` is between the current label and clearance of the `LIO` computation that creates it. Function `putLMVar` fills an `LMVar l a` with a value of type `a` if it is

empty and blocks otherwise. Dually, `takeLMVar` empties an `LMVar l a` if it is full and blocks otherwise.

Note that both `takeLMVar` and `putLMVar` check if the `LMVar` is empty in order to proceed to modify its content. Precisely, `takeLMVar` and `putLMVar` perform a read and a write of the mutable location. Consequently, from a security point of view, operations on a given `LMVar l a` are executed only when the label `l` is below or equal to the clearance (i.e., $l \sqsubseteq C_{\text{cur}}$ due to the read) and above or equal to the current label (i.e., $L_{\text{cur}} \sqsubseteq l$ due to the write). Moreover, after either operation, L_{cur} is raised to l.

Many communication channels used in practice are often *bi-directional*, i.e., a read produces a write (and vice versa). For instance, reading a file may modify the access time in the inode; writing to a socket may produce an observable error if the connection is closed, etc. As described above, `LMVar` are bi-directional channels. If we were to treat them as uni-directional, observe that, a termination leak would be possible: a thread, whose current label is `Low` can use a `LMVar` labeled `Low` to send information to a computation whose current label is `High`; the `High` thread can then decide to empty the `LMVar` according to a secret value and thus leak information to the `Low` thread.

5 The external timing covert channel

In a real-world scenario IFC applications interact with unlabeled, or publicly observable, resources. For example, a server-side IFC web application interacts with a browser, which may itself be IFC-unaware, over a public network channel. Consequently, an adversary can take measurements *external* to the application (e.g., the application response time) from which they may infer information about confidential data computed by the application. Although our results generalize (e.g., to the storage covert channel), in this section we address the *external timing covert channel*: an application can leak information over a public channel to an observer that precisely measures message-arrival timings. Note that the content of a message does not need to be public (hence why the channel is considered *covert*); this is the case in a web application where a message may be encrypted with SSL, but the actual placement of a message on the channel is observable by a network attacker.

Most of the language-based IFC techniques that consider external timing channels are limited. Despite the successful use of external timing attacks to leak information in web [7, 17] and cryptographic [18, 26, 51] applications, they remain widely unaddressed by mainstream, practical IFC tools, including Jif [34]. Furthermore, most techniques that provide IFC in the presence of the external timing channel [1, 5, 20] are overly restrictive, e.g., they do not allow folding over secret data.

5.1 Mitigating the external timing channel

Recently, a predictive black-box mitigation technique for external timing channels has been proposed [3, 53]. The predictive mitigation technique assumes that the attacker has control of the application (which computes on secret data) and can measure the time a message is placed on a channel (e.g., when a response is sent to the browser). Treating the application as a black-box source of events, a mitigator is interposed between the application and the system output.

Internally, the mitigator keeps a *schedule* describing when outputs are to be produced. For example, the time mitigator might keep a schedule "predicting" that an output is to be produced every 1ms. If the application delivers events according to the schedule, or at a higher rate, the mitigator will be able to produce an output at every 1ms interval, according to the schedule, and thus leak no information.

Of course, the application may fail to deliver an event to the mitigator on time, and thus render the mitigator's schedule prediction false. At this point, the mitigator must handle the misprediction by selecting, or "predicting", a new schedule for the application. In most cases, this corresponds to doubling the application's *quantum*. For instance, following a misprediction of a quantum of 1 ms, an application will be then expected to produce an output every 2 ms. It is at the point of switching schedules where an attacker learns information: rather than seeing events spaced at 1 ms intervals, the attacker now observes outputs at 2 ms intervals, indicating that the application violated the predicted behavior (a decision that can be affected by secret data). However, Askarov et al. [3] show that the amount of information leaked by this *slow-doubling* mitigator is polylogarithmic in the application runtime.

Furthermore, the aspects of the predictive mitigation technique of [3, 53] that makes it particularly attractive for use in LIO are:

- The mitigator can adaptively reduce the quantum, as to increase the throughput of a well-behaved application in a manner that bounds the covert channel bandwidth (though the leakage factor is still greater than that of the slow-doubling mitigator);
- The mitigator can leverage public factors to decide a schedule. For example, in a web application setting where responses are mitigated, the arrival of an HTTP request can be used as a "reset" event. This is particularly useful as a quiescent application would otherwise be penalized (by increasing its quantum) for not producing an output according to the predicted schedule. Our web application of Section 8 implements this mitigation technique
- The amount of information leaked is bound by a combinatorial analysis on the number of observations an attacker can perform.

Monadic approach to black-box mitigation Pure functional programming languages, such as Haskell, are particularly suitable for mitigating external timing covert channels. Specifically, the use of monads for enforcing an evaluation-order and introducing side-effects allows for the reasoning and control of output events. Among many others, LIO is an example library that leverages this property of monads; `LIO` is simply a monad that performs side-effects according to IFC.

The functionality of different monads, such as I/O and error handling, can be combined in a modular fashion using *monad transformers* [31]. A monad transformer t, when applied to a monad m, generates a new, combined monad $t\ m$, that shares the behavior of monad m as well as the behavior of the monad encoded in the monad transformer. The modularity of monad transformers comes from the fact that they consider the underlying monad m opaque, i.e., the behavior of the monad transformer t does not depend on the internal structure of m. In this light, we adopt Zhang et al.'s system-oriented predictive black-box mitigator to a language-based security setting in the form of a monad transformer.

5.2 Language-based mitigators

We envision the implementation of mitigators that address covert channels other than external timing. For example, our ongoing work includes the implementation of a storage mitigator that addresses attacks which vary message (packet) length to encode secret information. Hence, our mitigation monad transformer `MitM s q` is polymorphic in the mitigator-specific state `s` and quantum type `q`:

```
newtype MitM s q m a = MitM ...
```

The time-mitigated monad transformer is a special case:

```
type TimeMitM = MitM TStamp TStampDiff
```

where the internal state `TStamp` is a time stamp, and the quantum `TStampDiff` is a time difference. Superficially, a value of type `TimeMitM m a` is a computation that produces a value of type `a`. Internally, a time measurement is taken whenever an output is to be emitted in the underlying monad `m`, the internal state and quantum are adjusted to reflect the event, and the output is delayed if it was produced ahead of the predicted schedule.

We provide the function `evalMitM`, which takes an action of type `MitM s q m a` and returns an action of type `m a`, which when executed will mitigate the computation outputs. Observe that the monad transformer leaves the possibility to use (almost) any underlying monad `m`, not just `LIO` or `IO`; this makes the monad transformer approach to mitigation quite general.

Unfortunately, this generality comes with a trade-off: either every computation `m` is mitigated, or trustworthy programmers must define *what* objects they wish to mitigate and *how* to mitigate them. Given that the former design choice would not allow for distinguishing between inputs and outputs, we implemented the latter and more explicit mitigation approach.

To define *what* is to be mitigated (e.g., a file handle, a socket, a reference, etc.), we provide the data type:

```
data Mitigated s q a = Mitigated ...
```

For example, a time-mitigated I/O file handle is simply:

```
type TimeMitigated = Mitigated TStamp TStampDiff
type Handle = TimeMitigated IO.Handle
```

The use of `Mitigated` allows us to do mitigation at very fine grain level. Specifically, the monad transformer can be used to implement a mitigator for each `Mitigated` value (henceforth "handle"). This allows an application to write to multiple files, all of which are mitigated independently, and thus may be written to, at different rates[3]. It remains for us to address: *how* are the mitigators defined?

Mitigators are defined as instances of the type class `Mitigator`, which provides two functions:

```
class MonadConcur m => Mitigator m s q where
  -- | Create a Mitigated "handle".
  mkMitigated :: Maybe s   -- ^ Internal state
              -> q         -- ^ Quantum
              -> m a       -- ^ Handle constructor
              -> MitM s q m (Mitigated s q a)

  -- | Mitigate an operation
  mitigate :: Mitigated s q a  -- ^ Mitigated "handle"
           -> (a -> m ())      -- ^ Output computation
           -> MitM s q m ()
```

Firstly, we note the context `MonadConcur m` is used to impose the requirement that the underlying monad be an `IO`-like monad which allows forking new threads (as to separate the mitigator from the computation being mitigated) and operations on mutable `MVars` (which are internal to the `MitM` transformer). Secondly, we highlight the `mkMitigated` function, which is used to create a mitigated handle given an initial state, quantum, and underlying constructor. The default implementation of `mkMitigated` creates the mitigator state (internal to the transformer) corresponding to the handle. A simplified version of our `openFile` operation shows how `mkMitigated` is used:

[3] In cases where schedule mispredictions are common, it is important to implement the *l-grace* period policy of [53]. The policy states that when there are more than l mispredictions, the new scheduling should affect all mitigators.

```
openFile :: FilePath -> IOMode -> TimeMitigated IO Handle
openFile f mode = mkMitigated Nothing q $ do
  h <- IO.openFile f mode   -- Handle constructor
  return h
    where q = mkQuant 1000 -- Initial quantum of 1ms
```

Here, the constructor `IO.openFile` creates a file handle to the file at path `f`. This constructor is supplied to `mkMitigated`, in addition to the "empty" state `Nothing`, and initial quantum `q` of 1 ms, which creates the corresponding mitigator and `Mitigated` handle (recall `Handle` is a type alias to `TimeMitigated IO.Handle`). We note that although the default definition of `mkMitigated` creates a mitigator per handle, instances may provide a definition that is more coarse-grained (e.g., associate mitigator with current thread).

Finally, each mitigator provides a definition for `mitigate`, which specifies how a computation should be mitigated. The function takes two arguments: the mitigated handle and a computation that produces an output on the handle. Our time mitigator instance

```
instance ... => Mitigator m TStamp TStampDiff where
  mitigate mh act = ...
```

provides a definition for `mitigate`. The action first retrieves the internal state of the mitigator corresponding to the mitigated handle `mh` and forks a new thread (allowing other mitigated actions to be executed). In the new thread, a time measurement t_1 is taken. Then, if the time difference between t_1 and the mitigator time stamp t_0 exceeds the quantum q, the new mitigator quantum is set to $2q$; otherwise, the computation is "suspended" for $t_1 + t_0$ microseconds. Following, `act` is executed, and the internal timestamp is replaced with the current time. Using MVars, we force operations on the same handle to be sequential and thus follow the latest schedule.

Continuing the example, we can now define a function we wish to be mitigated:

```
hPut :: Handle -> ByteString -> TimeMitigated IO ()
hPut mH bs = mitigate mH (\h -> IO.hPut h bs)
```

If `hPut` is invoked according to schedule (at least every 1 ms), the actual output function `IO.hPut` is used to write the provided byte-strings every 1 ms. Conversely, if the function does not follow the predicted schedule, the quantum will be increased and write-throughput to the file will decrease. Of course, this does not affect the schedule on a different handle (until a large number of mispredictions occur).

Adapting an existing program to have mitigated outputs comes almost for free: a *trustworthy* programmer needs to define the constructor functions, such as `openFile`, and output functions, such as `hPut`,

Listing 4 Syntax for values, expressions, and types.

$$
\begin{array}{rrl}
\text{Label:} & & l \\
\text{LMVar:} & & m \\
\text{Value:} & v ::= & \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid l \mid m \mid x \mid \lambda x.e \mid \texttt{fix}\, e \\
& & \mid \texttt{Lb}\, l\, e \mid (e)^{\texttt{LIO}} \mid \boxdot \mid \texttt{R}\, m \mid \bullet \\
\text{Expression:} & e ::= & v \mid e\, e \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{let}\, x = e\, \texttt{in}\, e \\
& & \mid \texttt{return}\, e \mid e \gg\!= e \mid \texttt{label}\, e\, e \\
& & \mid \texttt{unlabel}\, e \mid \texttt{lowerClr}\, e \mid \texttt{getLabel} \\
& & \mid \texttt{getClearance} \mid \texttt{labelOf}\, e \mid \texttt{out}\, e\, e \\
& & \mid \texttt{forkLIO}\, e\, e \mid \texttt{waitLIO}\, e \mid \texttt{newLMVar}\, e\, e \\
& & \mid \texttt{takeLMVar}\, e \mid \texttt{putLMVar}\, e\, e \mid \texttt{labelOfLMVar}\, e \\
\text{Type:} & \tau ::= & \texttt{Bool} \mid \texttt{()} \mid \tau \to \tau \mid \ell \mid \texttt{Labeled}\, \ell\, \tau \mid \texttt{Result}\, \ell\, \tau \\
& & \mid \texttt{LMVar}\, \ell\, \tau \mid \texttt{LIO}\, \ell\, \tau
\end{array}
$$

and simply *lift* all the remaining operations. Recall that `MitM` is a monad transformer, and thus we provide a definition for the function:

```
lift ::: Monad m => m a -> MitM s q m a
```

which lifts a computation in the `m` monad into the mitigation monad, without performing any actual mitigation. A simple example illustrating this is the definition of `hGet` which reads a specified number of bytes from a handle:

```
hGet :: Handle -> Int -> TimeMitigated IO ByteString
hGet mH = lift . IO.hGet . mitVal
```

where `mitVal` returns the handle encapsulated by `Mitigated`. It is worth noting that, although we focus on mitigating writing operations, in some systems a file read will be reflected in the file's inode `atime`, and thus should be also accordingly mitigated.

6 Formal semantics for LIO

In this section, we formalise our library for a simply typed Curry-style call-by-name λ-calculus with some extensions. Listing 4 defines the formal syntax for the language. Syntactic categories v, e, and τ represent values, expressions, and types, respectively. Values are side-effect free while expressions denote (possible) side-effecting computations. Due to lack of space, we only show the reduction and typing rules for the core part of the library. For more details, readers can refer to Appendix 1 available in the supplementary material.

Values The syntax category v includes the symbol `true` and `false` representing Boolean values. Symbol `()` represents the unit value. Symbol ℓ denotes security labels. Symbol m represents `MVar`s. Values include variables (x), functions ($\lambda x.e$), and recursive functions (`fix` e). Special syntax nodes are added to this category: `Lb` v e, $(e)^{\texttt{LIO}}$, `R` m, $\boxdot$, and $\bullet$. Node `Lb` v e denotes the run-time representation of a labeled value. Similarly, node $(e)^{\texttt{LIO}}$ denotes the run-time representation of a monadic `LIO` computation. Node $\boxdot$ denotes the run-time representation of an empty `MVar`. Node `R` m is the run-time representation of a handle, implemented as an `MVar`, that is used to access the result produced by spawned computations. Alternatively, `R` m can be thought of as an explicit *future*. Node $\bullet$ represents an erased term (explained in Section 7). None of these special nodes appear in programs written by users and they are merely introduced for technical reasons.

Expressions Expressions are composed of values (v), function applications (e e), conditional branches (`if` e `then` e `else` e), and local definitions (`let` $x = e$ `in` e). Additionally, expressions may involve operations related to monadic computations in the `LIO` monad. More precisely, `return` e and e »= e represent the monadic return and bind operations. Monadic operations related to the manipulation of labeled values inside the `LIO` monad are given by `label`, and `unlabel`. Expression `unlabel` e acquires the content of the labeled value e while in a `LIO` computation. Expression `label` e_1 e_2 creates a labeled value, with label e_1, of the result obtained by evaluating the `LIO` computation e_2. Expression `lowerClr` e allows lowering of the current clearance to e. Expressions `getLabel` and `getClearance` return the current label and current clearance of an `LIO` computation. Expression `labelOf` e obtains the security label of labeled values. Expression `out` e_1 e_2 denotes the output of e_2 to the output channel at security level e_1. For simplicity, we assume that there is only one output channel per security level. Expression `forkLIO` e_1 e_2 spawns a thread that computes e_2 and returns a labeled value with label e_1. Expression `waitLIO` e inspects the value returned by the spawned computation whose result is accessed by the handle e. Non-proper morphisms related to creating, reading, and writing labeled `MVar`s are respectively captured by expressions `newLMVar`, `takeLMVar`, and `putLMVar`.

Types We consider standard types for Booleans (`Bool`), unit (`()`), and function ($\tau \to \tau$) values. Type ℓ describes security labels. Type `Result` ℓ τ denotes handles used to access labeled results produced by spawned computations, where the results are of type τ and labeled with labels of type ℓ. Type `LMVar` ℓ τ describes labeled `MVar`s, with labels of type ℓ and storing values of type τ. Type `LIO` ℓ τ represents monadic `LIO` computations, with a result type τ and the security labels of type ℓ.

Listing 5 Typing rules for special syntax nodes.

$$\frac{}{\Gamma \vdash \bullet : \tau} \qquad \frac{}{\Gamma \vdash m : \texttt{LMVar}\ \ell\ \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Lb}\ l\ e : \texttt{Labeled}\ \ell\ \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e)^{\texttt{LIO}} : \texttt{LIO}\ \ell\ \tau} \qquad \frac{}{\Gamma \vdash \boxdot : \tau} \qquad \frac{\Gamma \vdash m : \texttt{LMVar}\ \ell\ \tau}{\Gamma \vdash \texttt{R}\ m : \texttt{Result}\ \ell\ \tau}$$

The typing judgments have the standard form $\Gamma \vdash e : \tau$, such that expression e has type τ assuming the typing environment Γ; we use Γ for both variable and store typings. Typing rules for the special syntax nodes are shown in Listing 5. These rules are liberal on purpose. Recall that special syntax nodes are run-time representations of certain values, e.g., labeled `MVar`s. Thus, they are only considered in a context where it is possible to uniquely deduce their types. The typing for the remaining terms and expressions are standard and we therefore do not describe them any further. We do not require any of the sophisticated features of Haskell's type-system, a direct consequence of the fact that security checks are performed at run-time. Since typing rules are straightforward, we assume that the type system is sound with respect to our semantics.

The `LIO` monad is essentially implemented as a State monad. To simplify the formalization and description of expressions, without loss of generality, we make the state of the monad part of the run-time environment. More precisely, each thread is accompanied by a local security run-time environment σ, which keeps track of the current label (σ.`lbl`) and clearence (σ.`clr`) of the running `LIO` computation. Common to every thread, the symbol Σ holds the global LMVar store ($\Sigma.\phi$) and the output channels ($\Sigma.\alpha_l$, one for every security label l). A store ϕ is a mapping from LMVars to labeled values, while an output channel is a queue of events of the form out(v) (output) or exit(v) (termination), for some value v. For simplicity, we assume that every store contains a mapping for every possible LMVar, which is initially the syntax node ($\bullet$). The run-time environments Σ, σ, and a `LIO` computation form a *sequential configuration* $\langle \Sigma, \langle \sigma, e \rangle \rangle$.

The relation $\langle \Sigma, \langle \sigma, e \rangle \rangle \xrightarrow{\alpha} \langle \Sigma', \langle \sigma', e' \rangle \rangle$ represents a single evaluation step from expression e, under the run-time environments Σ and σ, to expression e' and run-time environments Σ' and σ'. We define such relation in terms of a structured operational semantics via evaluation contexts [16]. We say that e reduces to e' in one step. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. Symbol α ranges over the *internal* events triggered by expressions (as illustrated in Listing 6 and explained below). We utilize internal events to communicate between the threads and the scheduler. Listing 6 shows the reductions rules for the

core contributions in our library. Rules (LAB) and (UNLAB) impose the same security constrains as for the sequential version of LIO [46]. Rule (LAB) generates a labeled value if and only if the label is between the current label and clearance of the LIO computation. Rule (UNLAB) requires that, when the content of a labeled value is "retrieved" and used in a LIO computation, the current label is raised ($\sigma' = \sigma[\texttt{lbl} \mapsto l']$, where $l' = \sigma.\texttt{lbl} \sqcup l$), thus capturing the fact that the remaining computation might depend on e. Output channels are treated as deques of events. We use a standard deque-like interface with operations ($\triangleleft$) and ($\triangleright$) for front and back insertion (respectively), and we also allow pattern-matching in the rules as a representation of deconstruction operations. Rule (OUTPUT) adds the event $\texttt{out}(v)$ to the end of the output channel at security level l ($\Sigma.\alpha_l \triangleright \texttt{out}(v)$).

The main contributions of our language are related to the primitives for concurrency and synchronization. Rule (LFORK) allows for the creation of a thread and generates the internal event $\texttt{fork}(e)$, where e is the computation to spawn. The rule allocates a new LMVar in order to store the result produced by the spawned thread ($e \gg\!= \lambda x.\texttt{putLMVar}\ m\ x$). Using that LMVar, the rule provides a handle to access to the thread's result ($\texttt{return}\ (\texttt{R}\ m)$). Rule (LWAIT) simply uses the LMVar for the handle. As mentioned in Section 4, operations on LMVar are *bi-directional* and consequently the rules (NLMVAR), (TLMVAR), and (PLMVAR) require not only that the label of the mentioned LMVar be between the current label and current clearance of the thread ($\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr}$), but that the current labe be raised appropriately. Considering the security level of a LMVar (l), rule (TLMVAR) accordingly raises the current label ($\sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l]$) when emptying ($\Sigma.\phi[m \mapsto \texttt{Lb}\ l\ \boxdot]$) its content ($\Sigma.\phi(m) = \texttt{Lb}\ l\ e$). Similarly, considering the security level of a LMVar (l), rule (PLMVAR) accordingly raises the current label ($\sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l]$) when filling ($\Sigma.\phi[m \mapsto \texttt{Lb}\ l\ e]$) its content ($\Sigma.\phi(m) = \texttt{Lb}\ l\ \boxdot$). Finally, rule (GLABR) fetches a labeled LMVar from the LMVar store ($e = \Sigma.\phi(m)$, i.e., a value of the form $\texttt{Lb}\ l\ m$), and returns its label.

Listing 7 shows the formal semantics for threadpools. The relation $\hookrightarrow$ represents a single evaluation step for the threadpool, in contrast with $\longrightarrow$ which is only for a single thread. We write $\hookrightarrow^*$ for the reflexive and transitive closure of $\hookrightarrow$. As mentioned, configurations are of the form $\langle \Sigma, t_s \rangle$, where Σ is the global runtime environment and t_s is a queue of sequential configurations. The front of the queue is the thread that is currently executing. Threads are scheduled in a round-robin fashion, like GHC. The thread at the front of the queue executes one step, and it is then moved to the back of the queue (see Rule (STEP)). If this step involves a fork (represented by $\overset{\text{fork}(e)}{\longrightarrow}$), a new thread is created at the back of the queue (see Rule (FORK)). Threads are also moved to the back of the threadpool if they are blocked, e.g., waiting to read a value from

an empty `LMVar` (see Rule (NO-STEP), we define $\not\longrightarrow$ as the impossibility to make any progress). When a thread finishes, i.e., it can no longer reduce, the final value is placed in the output channel indicated by the current label (σ.`lbl`), and the thread is removed from the queue (see Rule (EXIT)).

7 Security guarantees

In this section, we show that LIO computations have the property of termination-sensitive non-interference. As in [30, 38, 46], we prove this property by using the *term erasure* technique. The erasure function ε_L rewrites data at security levels that the attacker cannot observe into the syntax node $\bullet$.

Listing 8 defines the erasure function ε_L. This function is defined in such a way that $\varepsilon_L(e)$ contains no information above level L, i.e., the function ε_L replaces all the information more sensitive than L in e with a hole ($\bullet$). In most of the cases, the erasure function is simply applied homomorphically (e.g., $\varepsilon_L(e_1\ e_2) = \varepsilon_L(e_1)\ \varepsilon_L(e_2)$). For thread-pools, the erasure function is mapped into all sequential configurations; all threads with a current label above L are removed from the pool (filter $(\lambda\langle\sigma, e\rangle.e \not\equiv \bullet)$ (map $\varepsilon_L\ t_s$), where $\equiv$ denotes syntactic equivalence). The computation performed in a certain sequential configuration is erased if the current label is above L. For runtime environments and stores, we map the erasure function into their components. An output channel is erased into the empty channel (ϵ) if it is above L, otherwise the individual output events are erased according to ε_L. Similarly, a labeled value is erased if the label assigned to it is above L.

Following the definition of the erasure function, we introduce a new evaluation relation $\longrightarrow_L$ as follows:

$$\frac{\langle \Sigma, t_s\rangle \longrightarrow \langle \Sigma', t_s'\rangle}{\langle \Sigma, t_s\rangle \longrightarrow_L \varepsilon_L(\langle \Sigma', t_s'\rangle)}$$

The relation $\longrightarrow_L$ guarantees that confidential data, i.e., data not below level L, is erased as soon as it is created. We write $\longrightarrow_L^*$ for the reflexive and transitive closure of $\longrightarrow_L$. Similarly, we introduce a relation $\hookrightarrow_L$ as follows:

$$\frac{\langle \Sigma, t_s\rangle \hookrightarrow \langle \Sigma', t_s'\rangle}{\langle \Sigma, t_s\rangle \hookrightarrow_L \varepsilon_L(\langle \Sigma', t_s'\rangle)}$$

As usual, we write $\hookrightarrow_L^*$ for the reflexive and transitive closure of $\hookrightarrow_L$.

In order to prove non-interference, we will establish a simulation relation between $\hookrightarrow^*$ and $\hookrightarrow_L^*$ through the erasure function: erasing all secret data and then taking evaluation steps in $\hookrightarrow_L$ is equivalent to taking

steps in $\hookrightarrow$ first, and then erasing all secret values in the resulting configuration. Note that this relation would not hold if information from some level above L was being leaked by the program. In the rest of this section, we only consider well-typed terms to ensure there are no stuck configurations.

For simplicity, we assume that the space address of the memory store is split into different security levels and that allocation is deterministic. Therefore, the address returned when creating an LMVar with label l depends only on the LMVars with label l already in the store. For the sake of brevity, the proofs have been shortened in this section, but more details can be found in Appendix 1.

We start by showing that the evaluation relations $\longrightarrow_L$ and $\hookrightarrow_L$ are deterministic.

Proposition 1 (Determinacy of $\longrightarrow_L$). *If $\langle \Sigma, t\rangle \longrightarrow_L \langle \Sigma', t'\rangle$ and $\langle \Sigma, t\rangle \longrightarrow_L \langle \Sigma'', t''\rangle$, then $\langle \Sigma', t'\rangle = \langle \Sigma'', t''\rangle$.*

Proof. By induction on expressions and evaluation contexts, showing there is always a unique redex in every step.

Proposition 2 (Determinacy of $\hookrightarrow_L$). *If $\langle \Sigma, t_s\rangle \hookrightarrow_L \langle \Sigma', t'_s\rangle$ and $\langle \Sigma, t_s\rangle \hookrightarrow_L \langle \Sigma'', t''_s\rangle$, then $\langle \Sigma', t'_s\rangle = \langle \Sigma'', t''_s\rangle$.*

Proof. By induction on expressions and evaluation contexts, showing there is always a unique redex in every step and using Proposition 1.

The next lemma establishes a simulation between $\hookrightarrow^*$ and $\hookrightarrow^*_L$.

Lemma 1 (Many-step simulation). *If $\langle \Sigma, t_s\rangle \hookrightarrow^* \langle \Sigma', t'_s\rangle$, then $\varepsilon_L(\langle \Sigma, t_s\rangle) \hookrightarrow^*_L \varepsilon_L(\langle \Sigma', t'_s\rangle)$.*

Proof. In order to prove this result, we rely on properties of the erasure function, such as the fact that it is idempotent and homomorphic to the application of evaluation contexts and substitution. We show that the result holds by case analysis on the rule used to derive $\langle \Sigma, t_s\rangle \hookrightarrow^* \langle \Sigma', t'_s\rangle$, and considering different cases for threads whose current label is below (or not) level L. For more details, see Appendix 1.

The L-equivalence relation $\approx_L$ is an equivalence relation between configurations (and their parts), defined as the equivalence kernel of the erasure function ε_L: $\langle \Sigma, t_s\rangle \approx_L \langle \Sigma', r_s\rangle$ iff $\varepsilon_L(\langle \Sigma, t_s\rangle) = \varepsilon_L(\langle \Sigma', r_s\rangle)$. If two configurations are L-equivalent, they agree on all data below or at level L, i.e., they cannot be distinguished by an attacker at level L. Note that two queues are L-equivalent iff the threads with current label no higher than L are pairwise L-equivalent in the order that they appear in the queue.

The next theorem shows the non-interference property. It essentially states that if we take two executions of a program with two L-equivalent inputs, then for every intermediate step of the computation of the first run, there is a corresponding step in the computation of the second run which results in an L-equivalent configuration. Note that this also includes the termination channel, since L-equivalence of configurations requires that output channels have matching events, and termination is modelled as a special kind of output event.

Theorem 1 (Termination-sensitive non-interference). *Given a computation* e *(with no* `Lb`*,* $()^{\texttt{LIO}}$*,* $\boxdot$*,* `R`*, and* $\bullet$*) where* $\Gamma \vdash e : \texttt{Labeled}\ \ell\ \tau \rightarrow \texttt{LIO}\ \ell\ (\texttt{Labeled}\ \ell\ \tau')$*, an attacker at level* L*, an initial securiy context* σ*, and runtime environments* Σ_1 *and* Σ_2 *where* $\Sigma_1.\phi = \Sigma_2.\phi = \varnothing$ *and* $\Sigma_1.\alpha_k = \Sigma_2.\alpha_k = \epsilon$ *for all levels* k*, then*

$$\begin{aligned} &\forall e_1 e_2.(\Gamma \vdash e_i : \texttt{Labeled}\,\ell\,\tau)_{i=1,2} \wedge\ e_1 \approx_L e_2 \\ &\qquad \wedge \langle \Sigma_1, \langle \sigma, e\ e_1 \rangle \rangle \hookrightarrow^* \langle \Sigma_1', t_s^1 \rangle \\ &\Rightarrow \exists \Sigma_2' t_s^2.\langle \Sigma_2, \langle \sigma, e\ e_2 \rangle \rangle \hookrightarrow^* \langle \Sigma_2', t_s^2 \rangle \wedge\ \langle \Sigma_1', t_s^1 \rangle \approx_L \langle \Sigma_2', t_s^2 \rangle \end{aligned}$$

Proof. The result follows by combining Lemma 1 and Proposition 2 (Determinacy).

8 Example Application: Dating Website

In this section we evaluate the feasibility of leaking information through timing-based covert channels as well as the effectiveness and expressiveness of our extensions to LIO.

We built a simple dating website that allows third-party developers to build applications that interact with a common database. Our website exposes a shared key-value store to third-party apps encoding interested-in relationships. A key correspond to a user ID and its associated value represent the users that he/she is interested in. For simplicity, we do not consider the list of users sensitive, but interested-in relationships should remain confidential. In particular, a user should be able to learn which other users are interested in them, but should not be able to learn the interested-in relationships of other users.

The website consists of two main components: 1) a trusted web server that executes apps written using LIO and 2) untrusted third-party apps that may interact with users and read and write to the database. The database is simply a list of tuples mapping keys (users) to `LMVar`s storing lists of users. Apps are separated from each other by URL prefixes. For example, the URL `http://xycombinator.biz/App1` points to `App1`. Requests with a particular app's URL prefix are serviced by invoking the app's request handler in an IFC-constrained, and time-mitigated, environemt. We assume a powerful, but realistic adversary. In par-

ticular, malicious application writers may themselves be users of the dating site. We now consider the effectiveness of termination and timing channels in leaking the database.

Termination covert channel As detailed in Section 3, the implementation of LIO [46], with `toLabeled`, is susceptible to a termination channel attack. In the context of our dating-website, a malicious application $term$, running on behalf of an (authenticated) user a can be used to leak information on another (target) user t as follows:

- Authenticated adversary a issues a request that contains a guess that user t has an interest in g: `GET /`$term$`?target=`t`&guess=`g
- The trusted app container invokes the app $term$ and forwards the request to it.
- The application $term$ then executes the following LIO code:

```
toLabeled ⊤ $ do v <- lookupDB t
                 if g == v then ⊥ else return ()
return $ mkHtmlResp200 "Bad guess"
```

  Here, `lookupDB` t is used to perform a database lookup with key t. If g is present in the database entry, the app will not terminate, otherwise it will respond, denoting the guess was wrong.

We found the termination attack to be very effective. Specifically, we measured the time required to reconstruct a database of 10 users to be 73 seconds[4].

If `toLabeled` is prohibited and `forkLIO` is used instead, the termination attack cannot be mounted. This is because `waitLIO` first raises the label of the app request handler. An attempt to output a response to the client browser will not succeed since the current label of the handler cannot flow to the label of the client's browser. It is important to note that errors of this kind are made indistinguishable from non-terminating requests. To accomplish this, our dating site catches label violation errors and converts them to `⊥`.

Internal timing covert channel To carry out an internal timing attack, an app must execute two threads that share a common resource. Concretely, an app can use internal timing to leak information on a target user t as follows:

- Authenticated adversary a issue a request containing a guess that t is interested-in g: `GET /`$internal$`?target=`t`&guess=`g
- The trusted app container invokes the app $internal$.
- App $internal$ then executes the following LIO code:

[4] All our measurements were conducted on a laptop with a Intel Core i7 2620M (2.7GHz) processor and 8GB of RAM, with GHC 7.4.1.

```
varHigh <- fork $
  toLabeled T $ do
    v <- lookupDB t
    if g == v then sleep 5000 else return ()
  appendToAppStorage g
varLow <- fork $ do sleep 3000
                    appendToAppStore -1
wait varHigh
wait varLow
r <- readFromAppStore
return $ mkHtmlResp200 r
```

The code spawns two threads. The first reads the high value in a `toLabeled` then outputs the guess to a low-label store, however, if the guess is correct, it sleeps for five seconds before outputting the guess. The second thread simply outputs a place holder after waiting for three seconds. The result is that the ordering of outputs reveals whether the guess is correct. If the guess is incorrect, the store will read `g,-1`; if the guess is correct, the store will read `-1,g`.

We implemented a magnified version of the attack above by sending several requests to the server. The adversary repeatedly sends requests to *internal* for each user in the system as a guess g. As with the termination channel attack, we found that internal timing attack is feasible. For a database of 10 users we managed to recover the entries in 66.92 seconds.

Our modifications to LIO can be used to address the internal timing attacks described above; replacing `toLabeled` with `forkLIO` eliminates the internal timing leaks. More generally, we observe that by using `forkLIO`, the time when the app writes to the persistent storage (`appendToAppStore`) cannot be influenced by sensitive data. Similarly, replacing `fork` and `wait` by their LIO counterparts renders the attack futile.

External timing covert channel We consider a simple external timing attack to our dating website in which the adversary a has access to a high-precision timer. An app *external* colluding with a can use external timing to leak a target user t's interested-in relationship as follows:

- Authenticated adversary a issues requests containing the target user t: `GET /`*external*`?target=`t`&guess=`g
- The trusted container invokes *external* with the request.
- App *external* then proceeds to execute the following LIO code:

```
toLabeled T $ do
  v <- lookupDB t
  if g == v then sleep 5000 else return ()
return $ mkHtmlResp200 "done"
```

The attack is a component of the internal timing attack: given a target t and guess g, if the g was correct the thread sleeps; otherwise it does nothing. The attacker simply measures the response time – recognizing a delay as a correct guess.

Despite its simplicity, we also found this attack to be plausible. In 33 seconds, we recovered a database of 10 users. Addressing this attack we mitigated the app handler, as described in Section 5. The response time of an app is mitigated, taking into account the arrival of a request. Although we manged to recover 3 of the 10 user entries in 64 seconds—we found that recovering the remaining user entries was infeasible. Of course, the performance of well-behaved apps was unaffected.

9 Related Work

IFC security libraries The seminal work by Li and Zdancewic [29] presents an implementation of information-flow security as a library using a generalization of monads called Arrows [21]. Following this line of work, Tsai et al. [48] further consider side-effects and concurrency. Different from our approach, Tsai et al. provide termination-insensitive non-interference under a cooperative scheduler and no synchronization primitives. Russo et al. [38] eliminate the need for Arrows by showing an IFC security library based solely on monads. Their library leverages Haskell's type-system to statically enforce non-interference. Jaskelioff and Russo [23] propose a library that enforces non-interference by executing the program as many times as security levels, which is known as secure multi-execution [14]. Recently, Stefan et al. propose the use of the monad `LIO` to track information-flow dynamically [46]. Morgenstern et al. [32] encoded an authorization- and IFC-aware programming language in Agda. Their encoding, however, does not consider computations with side-effects. Devriese and Piessens [15] used monad transformers and parametrized monads [4] to enforce non-interference, both dynamically and statically. None of the approaches mentioned above deals with the termination channel. Moreover, none of them (except from Tsai et al.) handle concurrency.

Internal timing covert channel There are several approaches to deal with the internal timing covert channel. The work in [43–45, 50] relies on the non-realistic primitive `protect(c)` which, by definition, hides the timing behaviour of `c`. Our approach, on the other hand, relies on the fork primitive and the semantics for mutable locations. Assuming a scenario where it is possible to modify the scheduler, the work in [6, 35] propose a novel interaction between threads and the scheduler that is able to implement a generalized version of `protect(c)`. A series of work [22, 47, 52] prevents internal timing leaks by avoiding any races on public data.

Boudol and Castellani [8, 9] avoid internal timing leaks by disallowing public events after branching on secret data. The authors consider a fixed number of threads and no synchronization primitives. Russo and Sabelfeld [36] show how to remove internal timing leaks under a cooperative scheduling by manipulating yield commands. The termination channel is intrinsically present under cooperative scheduling, i.e., there is no way to decouple executions between threads. The work by Russo et al. [37] is the closest one to our approach to internal timing leaks. In that work, the authors introduce a code transformation, from a sequential program into a concurrent one, that spawns threads to execute branches and loops whose conditionals depend on secret values. The idea of spawning threads when computations use secrets is similar to ours, but it is used in a quite different context. Firstly, Russo et al. apply their technique for a simple sequential while-language, while we consider concurrent programs with synchronization primitives. Secondly, and different from our work, their approach does not consider leaks due to termination, i.e., the transformation guarantess termination-insensitive non-interference. Finally, incurring high synchronization costs, the code transformation introduces synchronization between spawned threads in order to preserve the semantics of the original sequential program. The transformation might change the terminating behavior of programs in order to preserve security. Our proposal, on the other hand, guarantees that the semantics of the program is the one that the programmer writes in the code.

Termination and external covert channels There are several language-based mechanisms to tackle the termination and external timing channels. Volpano [49] describes a type-system that removes the termination channel by forbidding loops whose conditional depend on secrets. The work by Hedin and Sands [20] avoids the termination and external timing covert channels for sequential Java bytecode by disallowing outputs after branching on secrets. Similarly, `LIO` computations do not allow public outputs after observing secret data. However, the programmer can spawn new threads to perform such computations and thus allowing the rest of the system to still perform public outputs. Agat [1] describes a code transformation that removes external timing leaks by padding programs with dummy computations. The termination channel is closed by disallowing loops on secrets. One drawback of Agat's transformation is that if there is an if-then-else, whose guard depends on secret data, and only one of its branches is non-terminating, then the transformed program becomes non-terminating. This approach has been adapted for languages with concurrency [39–41]. Moreover, the transformation has been rephrased as a unification problem [27] as well as being implemented using transactions [5]. While targeting sequential programs, secure multi-execution [14] removes both the termination and external

timing channel. However, the latter is only closed if there are as many CPUs (or cores) as security levels being considered by the technique. We refer the reader to [25] for a more detailed description of possible enforcements for timing- and termination-sensitive non-interference. Recently, Zhang et al. [54] propose a language-based mitigation approach for a simple while-language extended with a `mitigate` primitive. Their work relies on static annotations to provide information about the underlying hardware. Compared to their work, our functional approach is more general and can be extended to address other covert channels (e.g., storage). However, their attack model is more powerful in considering the effects of hardware (e.g., cache). Nevertheless, we find their work to be complimentary: our system can leverage static annotations and the Xenon "no-fill" mode to address attacks relying on underlying hardware.

10 Summary

Many information flow control systems allow applications to sequence code with publicly visible side-effects after code computing over private data. Unfortunately, such sequencing leaks private data through termination channels (which affect whether the public side-effects ever happen), internal timing channels (which affect the order of publicly visible side-effects), and external timing channels (which affect the response time of visible side-effects). Such leaks are far worse in the presence of concurrency, particularly when untrusted code can spawn new threads.

We demonstrate that such sequencing can be avoided by introducing additional concurrency when public values must reference the results of computations over private data. We implemented this idea in an existing Haskell information flow library, LIO. In addition, we show how our library is amenable to mitigating external timing attacks by quantizing the appearance of externally visible side-effects. To evaluate our ideas, we prototyped the core of a dating web site showing that our interfaces are practical and our implementation does indeed mitigate these covert channels.

Listing 6 Semantics for non-standard expressions.

$$
\begin{aligned}
E \quad ::= \; & \ldots \mid \texttt{label}\ E\ e \mid \texttt{unlabel}\ E \mid \texttt{out}\ E\ e \mid \texttt{out}\ l\ E \\
& \mid \texttt{forkLIO}\ E\ e \mid \texttt{newLMVar}\ E\ e \mid \texttt{takeLMVar}\ E \\
& \mid \texttt{putLMVar}\ E\ e \mid \texttt{labelOfLMVar}\ E
\end{aligned}
$$

(LAB)
$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr}}{\langle \Sigma, \langle \sigma, E[\texttt{label}\ l\ e] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{return}\ (\texttt{Lb}\ l\ e)] \rangle \rangle}$$

(UNLAB)
$$\frac{l' = \sigma.\texttt{lbl} \sqcup l \qquad l' \sqsubseteq \sigma.\texttt{clr} \qquad \sigma' = \sigma[\texttt{lbl} \mapsto l']}{\langle \Sigma, \langle \sigma, E[\texttt{unlabel}\ (\texttt{Lb}\ l\ e)] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma', E[\texttt{return}\ e] \rangle \rangle}$$

(OUTPUT)
$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\alpha_l \mapsto \Sigma.\alpha_l \triangleright \texttt{out}(v)]}{\langle \Sigma, \langle \sigma, E[\texttt{out}\ l\ v] \rangle \rangle \longrightarrow \langle \Sigma', \langle \sigma, E[\texttt{return}\ ()] \rangle \rangle}$$

(LFORK)
$$\frac{\begin{array}{c}\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ \boxdot]] \\ \alpha = e \gg\!= \lambda x.\texttt{putLMVar}\ m\ x \qquad m \textit{ fresh}\end{array}}{\langle \Sigma, \langle \sigma, E[\texttt{forkLIO}\ l\ e] \rangle \rangle \overset{\texttt{fork}(e)}{\longrightarrow} \langle \Sigma', \langle \sigma, E[\texttt{return}\ (\texttt{R}\ m)] \rangle \rangle}$$

(LWAIT)
$$\langle \Sigma, \langle \sigma, E[\texttt{waitLIO}\ (\texttt{R}\ m)] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{takeLMVar}\ m] \rangle \rangle$$

(NLMVAR)
$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ e]] \qquad m \textit{ fresh}}{\langle \Sigma, \langle \sigma, E[\texttt{newLMVar}\ l\ e] \rangle \rangle \longrightarrow \langle \Sigma', \langle \sigma, E[\texttt{return}\ m] \rangle \rangle}$$

(TLMVAR)
$$\frac{\begin{array}{c}\Sigma.\phi(m) = \texttt{Lb}\ l\ e \qquad \sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \\ \sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l] \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ \boxdot]]\end{array}}{\langle \Sigma, \langle \sigma, E[\texttt{takeLMVar}\ m] \rangle \rangle \longrightarrow \langle \Sigma', \langle \sigma', E[\texttt{return}\ e] \rangle \rangle}$$

(PLMVAR)
$$\frac{\begin{array}{c}\Sigma.\phi(m) = \texttt{Lb}\ l\ \boxdot \qquad \sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \\ \sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l] \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ e]]\end{array}}{\langle \Sigma, \langle \sigma, E[\texttt{putLMVar}\ m\ e] \rangle \rangle \longrightarrow \langle \Sigma', \langle \sigma', E[\texttt{return}\ ()] \rangle \rangle}$$

(GLABR)
$$\frac{e = \Sigma.\phi(m)}{\langle \Sigma, \langle \sigma, E[\texttt{labelOfLMVar}\ m] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{labelOf}\ e] \rangle \rangle}$$

Listing 7 Semantics for threadpools.

$$\text{(STEP)}\quad \frac{\langle \Sigma, t\rangle \longrightarrow \langle \Sigma', t'\rangle}{\langle \Sigma, t \triangleleft t_s\rangle \hookrightarrow \langle \Sigma', t_s \triangleright t'\rangle} \qquad \text{(NO-STEP)}\quad \frac{\langle \Sigma, t\rangle \not\longrightarrow}{\langle \Sigma, t \triangleleft t_s\rangle \hookrightarrow \langle \Sigma, t_s \triangleright t\rangle}$$

$$\text{(FORK)}\quad \frac{\langle \Sigma, t\rangle \xrightarrow{\texttt{fork}(e)} \langle \Sigma', \langle \sigma, e'\rangle\rangle \qquad t_{\text{new}} = \langle \sigma, e\rangle}{\langle \Sigma, t \triangleleft t_s\rangle \hookrightarrow \langle \Sigma', t_s \triangleright \langle \sigma, e'\rangle \triangleright t_{\text{new}}\rangle}$$

$$\text{(EXIT)}\quad \frac{l = \sigma.\texttt{lbl} \qquad \Sigma' = \Sigma[\alpha_l \mapsto \Sigma.\alpha_l \triangleright \texttt{exit}(v)]}{\langle \Sigma, \langle \sigma, v\rangle \triangleleft t_s\rangle \hookrightarrow \langle \Sigma', t_s\rangle}$$

Listing 8 Erasure function.

$$\varepsilon_L(\langle \Sigma, t_s\rangle) = \langle \varepsilon_L(\Sigma), \text{filter } (\lambda\langle \sigma, e\rangle.e \notin \bullet)\ (\text{map } \varepsilon_L\ t_s)\rangle$$

$$\varepsilon_L(\langle \sigma, e\rangle) = \begin{cases} \langle \sigma, \bullet\rangle & \sigma.\texttt{lbl} \notin L \\ \langle \sigma, \varepsilon_L(e)\rangle & \text{otherwise} \end{cases}$$

$$\varepsilon_L(\Sigma) = \Sigma[\phi \mapsto \varepsilon_L(\Sigma.\phi)][\alpha_l \mapsto \varepsilon_L(\alpha_l)]_{l \in \text{Labels}}$$

$$\varepsilon_L(\alpha_l) = \begin{cases} \epsilon & l \notin L \\ \text{map } \varepsilon_L\ \alpha_l & \text{otherwise} \end{cases}$$

$$\varepsilon_L(\phi) = \{(x, \varepsilon_L(\phi(x))) : x \in \text{dom}(\phi)\}$$

$$\varepsilon_L(\texttt{Lb}\ l\ e) = \begin{cases} \texttt{Lb}\ l\ \bullet & l \notin L \\ \texttt{Lb}\ l\ \varepsilon_L(e) & \text{otherwise} \end{cases}$$

In the rest of the cases, ε_L is homomorphic.

BIBLIOGRAPHY

[1] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, Jan. 2000.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. of the 13th ESORICS*. Springer-Verlag, 2008.

[3] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM CCS*, CCS '10. ACM, 2010.

[4] R. Atkey. Parameterised notions of computation. In *Workshop on mathematically structured functional programming, ed. Conor McBride and Tarmo Uustalu. Electronic Workshops in Computing, British Computer Society*, pages 31–45, 2006.

[5] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. *Electron. Notes Theor. Comput. Sci.*, 153, May 2006.

[6] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *Proc. European Symposium on Research in Computer Security (ESORICS)*, pages 2–18, Sept. 2007.

[7] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07. ACM, 2007.

[8] Boudol and Castellani. Noninterference for concurrent programs. In *Proc. ICALP'01*, volume 2076 of *LNCS*. Springer-Verlag, July 2001.

[9] G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1), June 2002.

[10] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12*, pages 1–1. USENIX Association, 2003.

[11] D. Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of Cryptology*, 10(4), 1997.

[12] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5): 236–243, May 1976.

[13] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[14] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10. IEEE Computer Society, 2010.

[15] D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *Proc. of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM, 2011.

[16] M. Felleisen. The theory and practice of first-class prompts. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, pages 180–190. ACM, 1988.

[17] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM conference on Computer and communications security*, CCS '00. ACM, 2000.

[18] H. Handschuh and H. M. Heys. A timing attack on RC5. In *Proc. of the Selected Areas in Cryptography*. Springer-Verlag, 1999.

[19] D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Proc. of the 2011 Marktoberdorf Summer School*. IOS Press, 2011.

[20] D. Hedin and D. Sands. Timing aware information flow security for a javacard-like bytecode. *Elec. Notes Theor. Comput. Sci.*, 141, 2005.

[21] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[22] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, July 2006.

[23] M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2011.

[24] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1996.

[25] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. of IEEE Symposium on Sec. and Privacy*. IEEE, 2011.

[26] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. of the 16th CRYPTO*. Springer-Verlag, 1996.

[27] B. Köpf and H. Mantel. Eliminating implicit information leaks by transformational typing and unification. In *Formal Aspects in Security and Trust, Third International Workshop (FAST'05)*, volume 3866 of *LNCS*. Springer-Verlag, July 2006.

[28] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[29] P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW '06: Proc. of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

[30] P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19): 1974–1994, 2010.

[31] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *In Proceedings of the 22nd ACM Symposium on Principles of Programming Languages. ACMPress*, 1995.

[32] J. Morgenstern and D. R. Licata. Security-typed programming within dependently typed programming. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2010.

[33] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 129–142, 1997.

[34] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[35] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 177–189, July 2006.

[36] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics (PSI)*, LNCS. Springer-Verlag, June 2006.

[37] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Proc. of Asian Computing Science Conference (ASIAN)*, LNCS. Springer-Verlag, Dec. 2006.

[38] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN Symposium on Haskell (HASKELL)*, pages 13–24. ACM Press, Sept. 2008.

[39] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.

[40] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, Sept. 2002.

[41] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 200–214, July 2000.

[42] V. Simonet. The Flow Caml system. Software release at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

[43] Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 3–13, 2003.

[44] G. Smith. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, June 2001.

[45] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[46] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

[47] T. Terauchi. A type system for observational determinism. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 287–300. IEEE Computer Society, 2008.

[48] T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. IEEE Computer Security Foundations Symposium (CSF)*, July 2007.

[49] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th IEEE workshop on Computer Security Foundations*, CSFW '97. IEEE Computer Society, 1997.

[50] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3), Nov. 1999.

[51] W. H. Wong. Timing attacks on RSA: revealing your secrets through the fourth dimension. *Crossroads*, 11, May 2005.

[52] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 29–43, June 2003.

[53] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proc. of the 18th ACM CCS*. ACM, 2011.

[54] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proc. of PLDI*. ACM, 2012.

1 Detailed proofs

In this section, we provide more details about the proofs for the results in Section 7 as well as some auxiliary lemmas.

The following lemmas are necessary to prove that $\longrightarrow_L^*$ and $\hookrightarrow_L^*$ are deterministic.

Proposition 3 (Determinacy of $\longrightarrow$). *If $\langle \Sigma, t_s \rangle \longrightarrow \langle \Sigma', t'_s \rangle$ and $\langle \Sigma, t_s \rangle \longrightarrow \langle \Sigma'', t''_s \rangle$, then $\langle \Sigma', t'_s \rangle = \langle \Sigma'', t''_s \rangle$.*

Proof. By induction on expressions and evaluation contexts, showing there is always a unique redex in every step.

Proposition 1 (Determinacy of $\longrightarrow_L$). *If $\langle \Sigma, t \rangle \longrightarrow_L \langle \Sigma', t' \rangle$ and $\langle \Sigma, t \rangle \longrightarrow_L \langle \Sigma'', t'' \rangle$, then $\langle \Sigma', t' \rangle = \langle \Sigma'', t'' \rangle$.*

Proof. By Proposition 3 and definition of ε_L.

Proposition 4 (Determinacy of $\hookrightarrow$). *If $\langle \Sigma, t_s \rangle \hookrightarrow \langle \Sigma', t'_s \rangle$ and $\langle \Sigma, t_s \rangle \hookrightarrow \langle \Sigma'', t''_s \rangle$, then $\langle \Sigma', t'_s \rangle = \langle \Sigma'', t''_s \rangle$.*

Proof. By induction on expressions and evaluation contexts, showing there is always a unique redex in every step, and using Proposition 3.

Proposition 2 (Determinacy of $\hookrightarrow_L$). *If $\langle \Sigma, t_s \rangle \hookrightarrow_L \langle \Sigma', t'_s \rangle$ and $\langle \Sigma, t_s \rangle \hookrightarrow_L \langle \Sigma'', t''_s \rangle$, then $\langle \Sigma', t'_s \rangle = \langle \Sigma'', t''_s \rangle$.*

Proof. By Proposition 4 and the definition of ε_L.

The following proposition shows that the erasure function is homomorphic to the application of evaluation contexts and substitution, and that it is idempotent.

Proposition 5 (Properties of erasure function).

1. $\varepsilon_L(E[e]) = \varepsilon_L(E)[\varepsilon_L(e)]$
2. $\varepsilon_L([e_2/x]e_1) = [\varepsilon_L(e_2)/x]\varepsilon_L(e_1)$
3. $\varepsilon_L(\varepsilon_L(e)) = \varepsilon_L(e)$
4. $\varepsilon_L(\varepsilon_L(E)) = \varepsilon_L(E)$
5. $\varepsilon_L(\varepsilon_L(\Sigma)) = \varepsilon_L(\Sigma)$
6. $\varepsilon_L(\varepsilon_L(\langle\sigma, e\rangle)) = \varepsilon_L(\langle\sigma, e\rangle)$
7. $\varepsilon_L(\varepsilon_L(t_s)) = \varepsilon_L(t_s)$
8. $\varepsilon_L(\varepsilon_L(\langle\!|\Sigma, t_s|\!\rangle)) = \varepsilon_L(\langle\!|\Sigma, t_s|\!\rangle)$

Proof. All follow from the definition of the erasure function ε_L, and by induction on expressions and evaluation contexts.

Most of the reduction rules in Listing 6 will change the runtime environment. In addition, these transformations usually depend on a given expression, e.g. $\Sigma \mapsto \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto e]]$ can be seen as a function of e. We will represent these runtime transformations as functions $h : e \times \Sigma \to \Sigma$, where e is the set of expressions and Σ is the set of runtime environments. We will also write $h_e : \Sigma \to \Sigma$ for the partial application of h to an expression e. We extend this notation to transformations of stores and output channels.

We say that a transformation $f : e \times A \to A$ is *L-independent* if the secrets introduced in structure A by the application of f_e cannot be observed by an attacker at level L, i.e.

$$\varepsilon_L \circ f_e = \varepsilon_L \circ f_{\varepsilon_L(e)} \circ \varepsilon_L.$$

The next lemma is useful in proving that a given environment transformation is L-independent, by showing that its corresponding store and output channel transformations are L-independent.

Lemma 2. *Let h_e be a transformation for runtime environments that depends on an expression e, given as $h_e(\Sigma) = \Sigma[\phi \mapsto f_e(\Sigma.\phi)][\alpha_l \mapsto g_l^e(\Sigma.\alpha_l)]$ and thus uniquely determined by functions f_e and g_l^e for every label l and expression e. If f and g_l are all L-independent, then h is L-independent.*

Proof.

$$\begin{aligned}
&\varepsilon_L(h_{\varepsilon_L(e)}(\varepsilon_L(\Sigma))) \\
&= \varepsilon_L(\varepsilon_L(\Sigma)[\phi \mapsto f_{\varepsilon_L(e)}(\varepsilon_L(\Sigma).\phi)] \\
&\qquad [\alpha_l \mapsto g_l^{\varepsilon_L(e)}(\varepsilon_L(\Sigma).\alpha_l)]) \\
&= \varepsilon_L(\varepsilon_L(\Sigma)[\phi \mapsto f_{\varepsilon_L(e)}(\varepsilon_L(\Sigma.\phi))] \\
&\qquad [\alpha_l \mapsto g_l^{\varepsilon_L(e)}(\varepsilon_L(\Sigma.\alpha_l))]) \\
&= \varepsilon_L(\varepsilon_L(\Sigma))[\phi \mapsto \varepsilon_L(f_{\varepsilon_L(e)}(\varepsilon_L(\Sigma.\phi)))] \\
&\qquad [\alpha_l \mapsto \varepsilon_L(g_l^{\varepsilon_L(e)}(\varepsilon_L(\Sigma.\alpha_l)))] \\
&= \varepsilon_L(\Sigma)[\phi \mapsto \varepsilon_L(f_e(\Sigma.\phi))][\alpha_l \mapsto \varepsilon_L(g_l^e(\Sigma.\alpha_l))] \\
&= \varepsilon_L(\Sigma)[\phi \mapsto f_e(\Sigma.\phi)][\alpha_l \mapsto g_l^e(\Sigma.\alpha_l)] \\
&= \varepsilon_L(h_e(\Sigma))
\end{aligned}$$

The next lemma shows that the environment transformations in the reduction rules are all L-independent.

Lemma 3. *All runtime transformations h_e in the reduction rules in Listing 6 are L-independent.*

Proof. There are two cases to consider: modifications to the store (ϕ), which only update the contents of one reference, or appending a value to an output channel.

- **Case** $h_e(\Sigma) = \Sigma[\phi \mapsto f_e(\Sigma.\phi)]$**, with** $f_e(\phi) = \phi[m \mapsto \texttt{Lb}\ l\ e]$. By Lemma 2, we only have to prove that f is L-independent. We consider two cases:
  - $l \sqsubseteq L$:
    $$\begin{aligned} & \varepsilon_L(f_{\varepsilon_L(e)}(\varepsilon_L(\phi))) \\ &= \varepsilon_L(\varepsilon_L(\phi)[m \mapsto \texttt{Lb}\ l\ \varepsilon_L(e)]) \\ &= \varepsilon_L(\varepsilon_L(\phi[m \mapsto \texttt{Lb}\ l\ e])) \\ &= \varepsilon_L(f_e(\phi)) \end{aligned}$$
  - $l \not\sqsubseteq L$:
    $$\begin{aligned} & \varepsilon_L(f_{\varepsilon_L(e)}(\varepsilon_L(\phi))) \\ &= \varepsilon_L(\varepsilon_L(\phi)[m \mapsto \texttt{Lb}\ l\ \varepsilon_L(e)]) \\ &= \varepsilon_L(\varepsilon_L(\phi))[m \mapsto \texttt{Lb}\ l\ \bullet] \\ &= \varepsilon_L(\phi)[m \mapsto \texttt{Lb}\ l\ \bullet] \\ &= \varepsilon_L(f_e(\phi)) \end{aligned}$$
- **Case** $h_e(\Sigma) = \Sigma[\alpha_l \mapsto g_l^e(\Sigma.\alpha_l)]$ **with** $g_l^e(\alpha) = \alpha \triangleright e$. By Lemma 2, we only have to prove that g_l is L-independent.
  $$\begin{aligned} & \varepsilon_L(g_l^{\varepsilon_L(e)}(\varepsilon_L(\alpha))) \\ &= \varepsilon_L(\varepsilon_L(\alpha) \triangleright \varepsilon_L(e)) \\ &= \varepsilon_L(\varepsilon_L(\alpha \triangleright e)) \\ &= \varepsilon_L(g_l^e(\alpha)) \end{aligned}$$
- The rest of the cases are similar.

The following lemma establishes a simulation between $\longrightarrow$ and $\longrightarrow_L$ when reducing the body of a thread whose current label is below or equal to level L.

Lemma 4 (Single-step simulation for public computations).
If $\langle \Sigma, \langle \sigma, t\rangle\rangle \longrightarrow \langle \Sigma', t'\rangle$ with $\sigma.\texttt{lbl} \sqsubseteq L$, then $\varepsilon_L(\langle \Sigma, \langle \sigma, t\rangle\rangle) \longrightarrow_L \varepsilon_L(\langle \Sigma', t'\rangle)$.

$$\begin{array}{ccc} \langle \Sigma, \langle \sigma, e\rangle\rangle & \longrightarrow & \langle \Sigma^1, t'\rangle \\ \downarrow{\scriptstyle \varepsilon_L} & & \downarrow{\scriptstyle \varepsilon_L} \\ \varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle\rangle) & \underset{L}{\longrightarrow} & \varepsilon_L(\langle \Sigma^1, t'\rangle) \end{array}$$

Proof. The proof is by case analysis on the rule used to derive $\langle \Sigma, \langle \sigma, t \rangle \rangle \longrightarrow \langle \Sigma', t' \rangle$. As shown in Lemma 3, all environment modifications are consistent with the simulation: erasing secret data and then modifying the environment with erased data is equivalent to modifying the environment and then erasing the secrets.

- Case $t = E[\texttt{forkLIO}\ l\ e]$
$$\begin{aligned}
&\varepsilon_L(\langle \Sigma, \langle \sigma, E[\texttt{forkLIO}\ l\ e] \rangle \rangle) \\
&= \langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(E)[\texttt{forkLIO}\ l\ \varepsilon_L(e)] \rangle \rangle \\
&\overset{\text{fork}(\alpha)}{\longrightarrow}_L \varepsilon_L(\langle \varepsilon_L(\Sigma^1), \langle \sigma, \varepsilon_L(E)[\texttt{return}\ ()\ ] \rangle \rangle) \\
&= \varepsilon_L(\langle \varepsilon_L(\Sigma^1), \varepsilon_L(\langle \sigma, E[\texttt{return}\ ()\ ] \rangle) \rangle) \\
&\quad \text{(by Lemma 3, } \varepsilon_L(\Sigma^1) = \varepsilon_L(\Sigma')\text{)} \\
&= \varepsilon_L(\langle \Sigma', \langle \sigma, E[\texttt{return}\ ()\ ] \rangle \rangle)
\end{aligned}$$
- Case $t = E[\texttt{out}\ l\ e]$
$$\begin{aligned}
&\varepsilon_L(\langle \Sigma, \langle \sigma, E[\texttt{out}\ l\ e] \rangle \rangle) \\
&= \langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(E)[\texttt{out}\ l\ \varepsilon_L(e)] \rangle \rangle \\
&\longrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma^1), \langle \sigma, \varepsilon_L(E)[\texttt{return}\ ()\ ] \rangle \rangle) \\
&= \varepsilon_L(\langle \varepsilon_L(\Sigma^1), \varepsilon_L(\langle \sigma, E[\texttt{return}\ ()\ ] \rangle) \rangle) \\
&= \varepsilon_L(\langle \Sigma', \langle \sigma, E[\texttt{return}\ ()\ ] \rangle \rangle)
\end{aligned}$$
- Case $t = E[\texttt{takeLMVar}\ m]$
$$\begin{aligned}
&\varepsilon_L(\langle \Sigma, \langle \sigma, E[\texttt{takeLMVar}\ m] \rangle \rangle) \\
&= \langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(E)[\texttt{takeLMVar}\ m] \rangle \rangle \\
&\longrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma^1), \langle \sigma', \varepsilon_L(E)[\texttt{return}\ \varepsilon_L(e)] \rangle \rangle)
\end{aligned}$$
  Note that now $\sigma'.\texttt{lbl} = l$. We consider two cases:
  - $l \sqsubseteq L$:
$$\begin{aligned}
&\varepsilon_L(\langle \varepsilon_L(\Sigma^1), \langle \sigma', \varepsilon_L(E)[\texttt{return}\ \varepsilon_L(e)] \rangle \rangle) \\
&= \varepsilon_L(\langle \Sigma', \langle \sigma', E[\texttt{return}\ e] \rangle \rangle)
\end{aligned}$$
  - $l \not\sqsubseteq L$:
$$\begin{aligned}
&\varepsilon_L(\langle \varepsilon_L(\Sigma^1), \langle \sigma', \varepsilon_L(E)[\texttt{return}\ \varepsilon_L(e)] \rangle \rangle) \\
&= \langle \varepsilon_L(\Sigma'), \langle \sigma', \bullet \rangle \rangle \\
&= \varepsilon_L(\langle \Sigma', \langle \sigma', E[\texttt{return}\ e] \rangle \rangle)
\end{aligned}$$
  In both cases, it follows that $\varepsilon_L(\Sigma^1) = \varepsilon_L(\Sigma')$ by Lemma 3.
- Trivially reduces to the $t = E[\texttt{takeLMVar}\ m]$ case.
- Case $t = E[\texttt{newLMVar}\ l\ e]$.
$$\begin{aligned}
&\varepsilon_L(\langle \Sigma, \langle \sigma, E[\texttt{newLMVar}\ l\ e] \rangle \rangle) \\
&= \langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(E)[\texttt{newLMVar}\ l\ e] \rangle \rangle \\
&\longrightarrow_L \varepsilon_L(\langle \Sigma^1, \langle \sigma, \varepsilon_L(E)[\texttt{return}\ m] \rangle \rangle) \\
&= \varepsilon_L(\langle \Sigma^1, \varepsilon_L(\langle \sigma, E[\texttt{return}\ m] \rangle) \rangle) \\
&= \varepsilon_L(\langle \Sigma', \langle \sigma, E[\texttt{return}\ m] \rangle \rangle)
\end{aligned}$$
- Case $t = E[\texttt{putLMVar}\ m\ e]$.
$$\begin{aligned}
&\varepsilon_L(\langle \Sigma, \langle \sigma, E[\texttt{putLMVar}\ m\ e] \rangle \rangle) \\
&= \langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(E)[\texttt{putLMVar}\ m\ \varepsilon_L(e)] \rangle \rangle \\
&\longrightarrow_L \varepsilon_L(\langle \Sigma^1, \langle \sigma, \varepsilon_L(E)[\texttt{return}\ ()\ ] \rangle \rangle) \\
&= \varepsilon_L(\langle \Sigma^1, \varepsilon_L(\langle \sigma, E[\texttt{return}\ ()\ ] \rangle) \rangle) \\
&= \varepsilon_L(\langle \Sigma', \langle \sigma, E[\texttt{return}\ ()\ ] \rangle \rangle)
\end{aligned}$$

- The rest of the cases are similar.

The following lemma establishes a simulation between $\hookrightarrow$ and $\hookrightarrow_L$ when reducing the body of a thread whose current label is below or equal to level L.

Lemma 5 (Single-step simulation for public computations). *If* $\langle \Sigma, \langle \sigma, t\rangle \lhd t_s\rangle \hookrightarrow \langle \Sigma', t'_s\rangle$ *with* $\sigma.\mathtt{lbl} \sqsubseteq L$*, then* $\varepsilon_L(\langle \Sigma, \langle \sigma, t\rangle \lhd t_s\rangle) \hookrightarrow_L \varepsilon_L(\langle \Sigma', t'_s\rangle)$.

$$\begin{array}{ccc} \langle \Sigma, \langle \sigma, e\rangle \lhd t_s\rangle & \hookrightarrow & \langle \Sigma', t'_s\rangle \\ \downarrow{\scriptstyle \varepsilon_L} & & \downarrow{\scriptstyle \varepsilon_L} \\ \varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle \lhd t_s\rangle) & \underset{L}{\hookrightarrow} & \varepsilon_L(\langle \Sigma', t'_s\rangle) \end{array}$$

Proof. The proof is by case analysis on the rule used to derive $\langle \Sigma, \langle \sigma, t\rangle \lhd t_s\rangle \hookrightarrow \langle \Sigma', t'_s\rangle$.

- Case (STEP). By Lemma 4, we know that $\varepsilon_L(\langle \Sigma, t\rangle) \longrightarrow_L \varepsilon_L(\langle \Sigma', t'\rangle)$, so $\langle \varepsilon_L(\Sigma), \varepsilon_L(t) \lhd \varepsilon_L(t_s)\rangle \hookrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma'), \varepsilon_L(t_s) \rhd \varepsilon_L(t')\rangle)$.
$$\begin{aligned} & \varepsilon_L(\langle \Sigma, t \lhd t_s\rangle) \\ &= \langle \varepsilon_L(\Sigma), \varepsilon_L(t) \lhd \varepsilon_L(t_s)\rangle \\ &\hookrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma'), \varepsilon_L(t_s) \rhd \varepsilon_L(t')\rangle) \\ &= \varepsilon_L(\langle \Sigma', t_s \rhd t'\rangle) \end{aligned}$$
- Case (NO-STEP).
$$\begin{aligned} & \varepsilon_L(\langle \Sigma, t \lhd t_s\rangle) \\ &= \langle \varepsilon_L(\Sigma), \varepsilon_L(t) \lhd \varepsilon_L(t_s)\rangle \\ &\hookrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma), \varepsilon_L(t_s) \rhd \varepsilon_L(t)\rangle) \\ &= \varepsilon_L(\langle \Sigma, t_s \rhd t\rangle) \end{aligned}$$
- Case (FORK).
$$\begin{aligned} & \varepsilon_L(\langle \Sigma, \langle \sigma, t\rangle \lhd t_s\rangle) \\ &= \langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(t)\rangle \lhd \varepsilon_L(t_s)\rangle \\ &\hookrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma'), \varepsilon_L(t_s) \rhd \langle \sigma, \varepsilon_L(t')\rangle \rhd t_{\text{new}}\rangle) \\ &= \langle \varepsilon_L(\Sigma'), \varepsilon_L(t_s \rhd \langle \sigma, t'\rangle \rhd t_{\text{new}})\rangle \\ &= \varepsilon_L(\langle \Sigma', t_s \rhd \langle \sigma, t'\rangle \rhd t_{\text{new}}\rangle) \end{aligned}$$
- Case (EXIT).
$$\begin{aligned} & \varepsilon_L(\langle \Sigma, \langle \sigma, v\rangle \lhd t_s\rangle) \\ &= \langle \varepsilon_L(\Sigma), \langle \sigma, \varepsilon_L(v)\rangle \lhd \varepsilon_L(t_s)\rangle \\ &\hookrightarrow_L \varepsilon_L(\langle \Sigma^1, \varepsilon_L(t_s)\rangle) \\ &= \varepsilon_L(\langle \Sigma', t_s\rangle) \end{aligned}$$

We can also show that initial and final configurations for any reduction steps taken from a thread above L are equal when erased.

Lemma 6. *If* $\langle \Sigma, \langle \sigma, e\rangle\rangle \longrightarrow \langle \Sigma^1, t'\rangle$ *with* $\sigma.\texttt{lbl} \notin L$*, then* $\varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle\rangle) = \varepsilon_L(\langle \Sigma^1, t'\rangle)$*, i.e.,*

$$\begin{array}{ccc} \langle \Sigma, \langle \sigma, e\rangle\rangle & \longrightarrow & \langle \Sigma^1, t'\rangle \\ \Big\downarrow{\scriptstyle \varepsilon_L} & & \Big\downarrow{\scriptstyle \varepsilon_L} \\ \varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle\rangle) & = & \varepsilon_L(\langle \Sigma^1, t'\rangle) \end{array}$$

Proof. Since $\varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle\rangle) = \langle \varepsilon_L(\Sigma), \langle \sigma, \bullet\rangle\rangle$, we only have to show that $\varepsilon_L(\Sigma) = \varepsilon_L(\Sigma^1)$, where Σ^1 is the modified environment after performing the reduction step. The proof is similar to L-independence for the simulation lemma: for an arbitrary environment transformation h_e, we have to prove that $\varepsilon_L \circ h_e = \varepsilon_L$.

- **Case** $h_e(\Sigma) = \Sigma[\phi \mapsto f_e(\Sigma.\phi)]$**, with** $f_e(\phi) = \phi[m \mapsto \texttt{Lb}\ l\ e]$. We prove that $\varepsilon_L \circ f_e = \varepsilon_L$.
  $$\begin{aligned} & \varepsilon_L(f_e(\phi)) \\ = {} & \varepsilon_L(\phi[m \mapsto \texttt{Lb}\ l\ e]) \\ = {} & \varepsilon_L(\phi[m \mapsto \texttt{Lb}\ l\ \bullet]) \\ = {} & \varepsilon_L(\phi) \end{aligned}$$
- **Case** $h_e(\Sigma) = \Sigma[\alpha_l \mapsto g_l^e(\Sigma.\alpha_l)]$ **with** $g_l^e(\alpha) = \alpha \triangleright e$. Analogous.

Lemma 7. *If* $\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle \hookrightarrow \langle \Sigma^1, t'_s\rangle$ *with* $\sigma.\texttt{lbl} \notin L$*, then* $\varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle) = \varepsilon_L(\langle \Sigma^1, t'_s\rangle)$*, i.e.,*

$$\begin{array}{ccc} \langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle & \hookrightarrow & \langle \Sigma^1, t'_s\rangle \\ \Big\downarrow{\scriptstyle \varepsilon_L} & & \Big\downarrow{\scriptstyle \varepsilon_L} \\ \varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle) & = & \varepsilon_L(\langle \Sigma^1, t'_s\rangle) \end{array}$$

Proof. We illustrate the proof in the case of rule (STEP). Let $\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle \longrightarrow \langle \Sigma^1, t_s \triangleright \langle \sigma, e'\rangle\rangle$, then

$$\begin{aligned} & \varepsilon_L(\langle \Sigma, \langle \sigma, e\rangle \triangleleft t_s\rangle) \\ = {} & \langle \varepsilon_L(\Sigma), \varepsilon_L(t_s)\rangle \\ = {} & \langle \varepsilon_L(\Sigma^1), \varepsilon_L(t_s)\rangle \\ = {} & \varepsilon_L(\langle \Sigma^1, t_s \triangleright \langle \sigma, e'\rangle\rangle) \end{aligned}$$

The other cases are similar.

The next lemma establishes a simulation between $\hookrightarrow^*$ and $\hookrightarrow^*_L$.

Lemma 1 (Many-step simulation). *If* $\langle \Sigma, t_s\rangle \hookrightarrow^* \langle \Sigma', t'_s\rangle$*, then* $\varepsilon_L(\langle \Sigma, t_s\rangle) \hookrightarrow^*_L \varepsilon_L(\langle \Sigma', t'_s\rangle)$.

Proof. In order to prove this result, we rely on properties of the erasure function, such as the fact that it is idempotent and homomorphic to the application of evaluation contexts and substitution.

The proof is by induction on the derivation of $\langle \Sigma, t_s \rangle \hookrightarrow^* \langle \Sigma', t'_s \rangle$. We consider a thread queue of the form $\langle \sigma, e \rangle \triangleleft r_s$, and suppose that $\langle \Sigma, \langle \sigma, e \rangle \triangleleft r_s \rangle \hookrightarrow \langle \Sigma^1, r'_s \rangle$ and $\langle \Sigma^1, r'_s \rangle \hookrightarrow^* \langle \Sigma', t'_s \rangle$ (otherwise the reduction is not making any progress, and the result is trivial).

- If $\sigma.\texttt{lbl} \sqsubseteq L$, the result follows by Lemma 5 and the induction hypothesis.
- If $\sigma.\texttt{lbl} \not\sqsubseteq L$, the result follows by Lemma 7 and the induction hypothesis.

We can now prove the non-interference theorem.

Theorem 2 (Termination-sensitive non-interference). *Given a computation e (with no* `Lb`*, $()^{\texttt{LIO}}$, $\boxdot$, `R`*, and $\bullet$) where $\Gamma \vdash e : \texttt{Labeled}\ \ell\ \tau \to \texttt{LIO}\ \ell\ (\texttt{Labeled}\ \ell\ \tau')$, an attacker at level L, an initial securiy context σ, and runtime environments Σ_1 and Σ_2 where $\Sigma_1.\phi = \Sigma_2.\phi = \varnothing$ and $\Sigma_1.\alpha_k = \Sigma_2.\alpha_k = \epsilon$ for all levels k, then*

$$\begin{aligned} \forall e_1 e_2.(\Gamma \vdash e_i : \texttt{Labeled}\ \ell\ \tau)_{i=1,2} \wedge\ & e_1 \approx_L e_2 \\ \wedge\ & \langle \Sigma_1, \langle \sigma, e\ e_1 \rangle \rangle \hookrightarrow^* \langle \Sigma'_1, t^1_s \rangle \\ \Rightarrow \exists \Sigma'_2 t^2_s.\langle \Sigma_2, \langle \sigma, e\ e_2 \rangle \rangle & \hookrightarrow^* \langle \Sigma'_2, t^2_s \rangle \wedge\ \langle \Sigma'_1, t^1_s \rangle \approx_L \langle \Sigma'_2, t^2_s \rangle \end{aligned}$$

Proof. Take $\langle \Sigma_1, \langle \sigma, e\ e_1 \rangle \rangle \hookrightarrow^* \langle \Sigma'_1, t^1_s \rangle$ and apply Lemma 1 to get $\varepsilon_L(\langle \Sigma_1, \langle \sigma, e\ e_1 \rangle \rangle) \hookrightarrow^*_L \varepsilon_L(\langle \Sigma'_1, t^1_s \rangle)$. We know this reduction only includes public ($\sqsubseteq L$) steps, so the number of steps is lower than or equal to the number of steps in the first reduction.

We can always find a reduction starting from $\varepsilon_L(\langle \Sigma_2, \langle \sigma, e\ e_2 \rangle \rangle)$ with the same number of steps as $\varepsilon_L(\langle \Sigma_1, \langle \sigma, e\ e_1 \rangle \rangle) \hookrightarrow^*_L \varepsilon_L(\langle \Sigma'_1, t^1_s \rangle)$, so by the Determinacy Lemma we have $\varepsilon_L(\langle \Sigma_2, \langle \sigma, e\ e_2 \rangle \rangle) \hookrightarrow^*_L \varepsilon_L(\langle \Sigma'_2, t^2_s \rangle)$. By Lemma 1 again, we get $\langle \Sigma_2, \langle \sigma, e\ e_2 \rangle \rangle \hookrightarrow^* \langle \Sigma'_2, t^2_s \rangle$ and therefore $\langle \Sigma'_1, t^1_s \rangle \approx_L \langle \Sigma'_2, t^2_s \rangle$.

2 Semantics and typing rules

Listings 9 and 10 show the missing typing rules for the calculus. Similarly, Listing 11 shows the reduction rules that were not included in Section 6.

3 Application: Mitigating attack on RSA

As in [3], to highlight the effectiveness of our mitigator implementation, we re-implement the timing attack on the OpenSSL 0.9.7 RSA implementation as originally presented in [10]. Compared to the previous

Listing 9 Typing rules for values.

$$\frac{}{\vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{Bool}} \qquad \frac{}{\vdash () : ()} \qquad \frac{}{\vdash l : \ell}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \texttt{fix}\, e : \tau}$$

dating-website scenario, in which a malicious app deliberately delayed computations, the covert timing channel in this case is present due to the non-trivial operations performed in a decryption. Hence, an attacker can recover an RSA key by repeatedly requesting the RSA oracle, which may be a web server using SSL, to decrypt different ciphertext messages.

Following [10], one can reveal the secret key indirectly, by recovering q and exposing the factorization of RSA modulus $N = pq$, for $q < p$. To do so, the attack proceeds as follows. Firstly, it guesses an initial value for q, named g, that is between $2^{\log_2 N/2}$ and $2^{\log_2 N/2+1}$, and plots the decryption times (in nanoseconds) of all the most significant 2-3 bits. The expected peak in the plot graph corresponds to our first approximation of q. Assuming that the most significant $i+1$ bits of q have been already recovered, we recover the ith bit according to:

- Set the $i + 1$ most significant bits (MSB) of g_i to the $i + 1$ recovered MSB of q, leaving the remaining bits unset.
- Let g_{hi} being the same as g_i but with the ith bit set.
- Measure the time to decrypt g_i, written t_1.
- Measure the time to decrypt g_{hi}, written t_2.
- Compute $\Delta =\mid t_2 + t_1 \mid$. If Δ is large, bit i of q is unset, otherwise it is set.

As in [3, 10], we overcome noise due to the operating system being a multi-user environment by repeating the decryption for g_i and g_{hi} numerous times (in our experiments, 7) and taking the median time difference. Additionally, to build a strong indicator for the bits of q, we take the time difference of decrypting a neighborhood of values $g_i, \ldots, g_i + n$ and the corresponding neighborhood of high values $g_{hi}, \ldots, g_{hi} + n$; in our experiments $n = 600$.

To evaluate our Haskell mitigator implementation with the RSA attack, we extended the HsOpenSSL package with bindings for the C OpenSSL RSA encryption and decryption functions. On a laptop with a Intel Core i7 2620M (2.7GHz) processor with 8GB of RAM, we built our extended Haskell OpenSSL library with GHC 7.2.1, linking it against the C OpenSSL 0.9.7 library. The attack against a "toy" 512-bit key is shown Figure 1. We only carried out the attack against the 256 MSBs as Coppersmith's algorithm can be used to recover the rest in an efficient manner [11]. As the figure shows, there is a clear distinction between

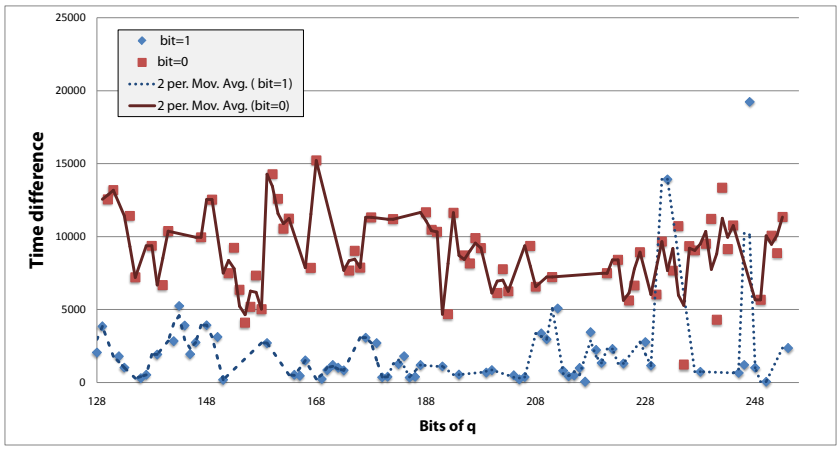

Fig. 1. Unmitigated RSA attack. Time difference is in nanoseconds.

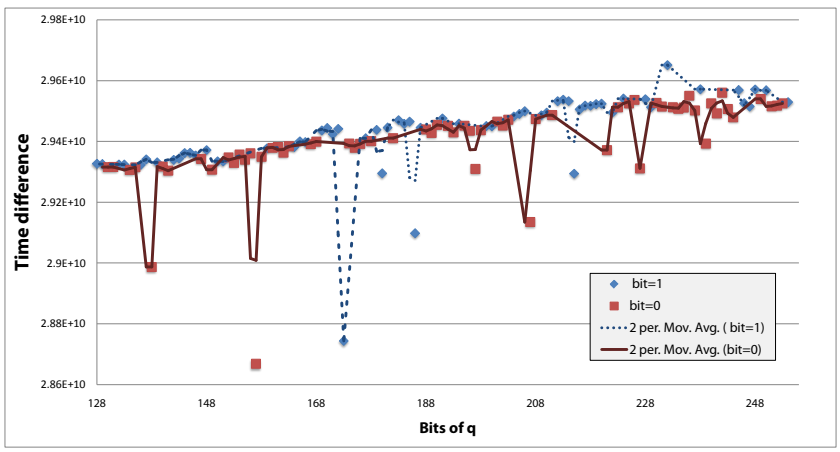

Fig. 2. Mitigated RSA attack. Time difference is in nanoseconds.

when the bits of q are 0 and 1. Finally, applying the fast-doubling time mitigator with an initial quantum of 500 microseconds, we bound the key leakage as shown by the results of Figure 2.

4 Evaluation: Overhead of a fork

To analyze the performance penalty in using `forkLIO` and `waitLIO` as opposed to `toLabeled` we micro-benchmarked the two approaches. As expected, Figure 3, the performance overhead of forking is unnoticeable.

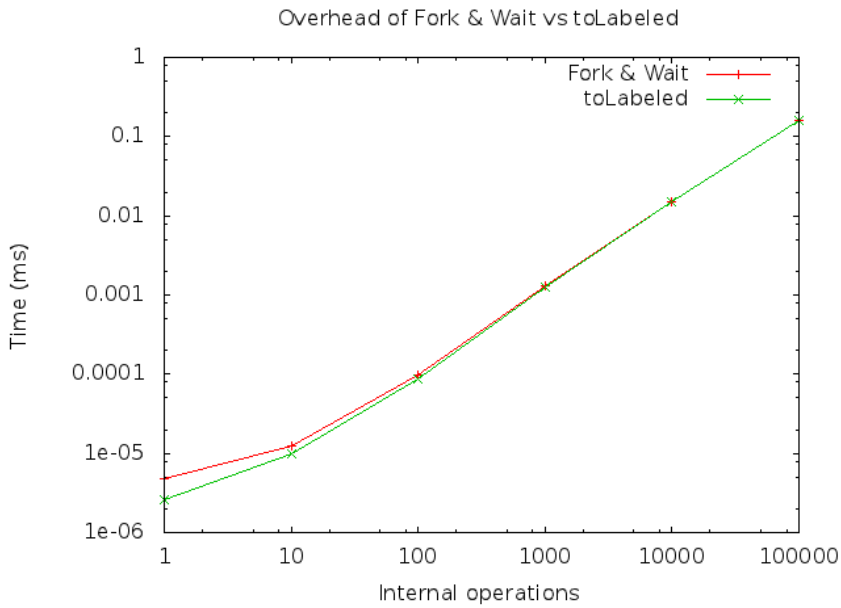

Fig. 3. Execution time in milliseconds for performing a `forkLIO` and `waitLIO` or `toLabeled`. The x-axis specifies the number of operations performed inside the `forkLIO` or `toLabeled`.

Listing 10 Typing rules for expressions.

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}\ e : \texttt{LIO}\ \ell\ \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{LIO}\ \ell\ \tau_1 \qquad \Gamma \vdash e_2 : \tau_1 \to \texttt{LIO}\ \ell\ \tau_2}{\Gamma \vdash e_1 \gg\!= e_2 : \texttt{LIO}\ \ell\ \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{label}\ e_1\ e_2 : \texttt{LIO}\ \ell\ (\texttt{Labeled}\ \ell\ \tau)} \qquad \frac{\Gamma \vdash e : \texttt{Labeled}\ \ell\ \tau}{\Gamma \vdash \texttt{unlabel}\ e : \texttt{LIO}\ \ell\ \tau}$$

$$\frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \texttt{LIO}\ \ell\ \tau}{\Gamma \vdash \texttt{forkLIO}\ e_1\ e_2 : \texttt{LIO}\ \ell\ (\texttt{Result}\ \ell\ \tau)} \qquad \frac{\Gamma \vdash e : \texttt{Result}\ \ell\ \tau}{\Gamma \vdash \texttt{waitLIO}\ e : \texttt{LIO}\ \ell\ \tau}$$

$$\frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{out}\ e_1\ e_2 : \texttt{LIO}\ \ell\ ()} \qquad \frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{newLMVar}\ e_1\ e_2 : \texttt{LIO}\ \ell\ (\texttt{LMVar}\ \ell\ \tau)}$$

$$\frac{\Gamma \vdash e : \texttt{LMVar}\ \ell\ \tau}{\Gamma \vdash \texttt{takeLMVar}\ e : \texttt{LIO}\ \ell\ \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{LMVar}\ \ell\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{putLMVar}\ e_1\ e_2 : \texttt{LIO}\ \ell\ ()}$$

$$\frac{\Gamma \vdash e : \ell}{\vdash \texttt{lowerClr}\ e : \texttt{LIO}\ \ell\ ()} \qquad \vdash \texttt{getLabel} : \texttt{LIO}\ \ell\ \ell$$

$$\vdash \texttt{getClearance} : \texttt{LIO}\ \ell\ \ell \qquad \frac{\Gamma \vdash e : \texttt{Lb}\ \ell\ \tau}{\Gamma \vdash \texttt{labelOf}\ e : \ell}$$

$$\frac{\Gamma \vdash m : \texttt{LMVar}\ \ell\ \tau}{\Gamma \vdash \texttt{labelOfLMVar}\ m : \ell}$$

Listing 11 Semantics for standard constructs.

$$
\begin{aligned}
E \quad & ::= [\cdot] \mid E\ e \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid \texttt{return}\ E \mid E \gg\!= e \\
& \mid \texttt{lowerClr}\ E \mid \texttt{labelOf}\ E \mid \ldots
\end{aligned}
$$

$$\langle \Sigma, \langle \sigma, E[(\lambda x.e_1)\ e_2] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[[e_2/x]e_1] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{fix}\ e] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[e\ (\texttt{fix}\ e)] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{if true then}\ e_1\ \texttt{else}\ e_2] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[e_1] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{if false then}\ e_1\ \texttt{else}\ e_2] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[e_2] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{let}\ x = e_1\ \texttt{in}\ e_2] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[[e_1/x]e_2] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{return}\ v] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[(v)^{\texttt{LIO}}] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[(v)^{\texttt{LIO}} \gg\!= e_2] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[e_2\ v] \rangle \rangle$$

$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr}}{\langle \Sigma, \langle \sigma, E[\texttt{label}\ l\ e] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{return}\ (\texttt{Lb}\ l\ e)] \rangle \rangle}$$

$$\frac{l' = \sigma.\texttt{lbl} \sqcup l \qquad l' \sqsubseteq \sigma.\texttt{clr} \qquad \sigma' = \sigma[\texttt{lbl} \mapsto l']}{\langle \Sigma, \langle \sigma, E[\texttt{unlabel}\ (\texttt{Lb}\ l\ e)] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma', E[\texttt{return}\ e] \rangle \rangle}$$

$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \sigma' = \sigma[\texttt{clr} \mapsto l]}{\langle \Sigma, \langle \sigma, E[\texttt{lowerClr}\ l] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma', E[\texttt{return}\ ()] \rangle \rangle}$$

$$\langle \Sigma, \langle \sigma, E[\texttt{getLabel}] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{return}\ \sigma.\texttt{lbl}] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{getClearance}] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{return}\ \sigma.\texttt{clr}] \rangle \rangle$$

$$\langle \Sigma, \langle \sigma, E[\texttt{labelOf}\ (\texttt{Lb}\ l\ e)] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[l] \rangle \rangle$$

CHAPTER
THREE

ELIMINATING CACHE-BASED TIMING ATTACKS WITH INSTRUCTION-BASED SCHEDULING

Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy
David Terei, Alejandro Russo, David Mazières

Abstract. Information flow control allows untrusted code to access sensitive and trustworthy information without leaking this information. However, the presence of covert channels subverts this security mechanism, allowing processes to communicate information in violation of IFC policies. In this paper, we show that concurrent deterministic IFC systems that use time-based scheduling are vulnerable to a cache-based internal timing channel. We demonstrate this vulnerability with a concrete attack on Hails, one particular IFC web framework. To eliminate this internal timing channel, we implement instruction-based scheduling, a new kind of scheduler that is indifferent to timing perturbations from underlying hardware components, such as the cache, TLB, and CPU buses. We show this scheduler is secure against cache-based internal timing attacks for applications using a single CPU. To show the feasibility of instruction-based scheduling, we have implemented a version of Hails that uses the CPU retired-instruction counters available on commodity Intel and AMD hardware. We show that instruction-based scheduling does not impose significant performance penalties. Additionally, we formally prove that our modifications to Hails' underlying IFC system preserve non-interference in the presence of caches.

1 Introduction

The rise of extensible web applications, like the Facebook Platform, is spurring interest in information flow control (IFC) [27, 35]. Popular platforms like Facebook give approved apps full access to users' sensitive data, including the ability to violate security policies set by users. In contrast, IFC allows websites to run untrusted, third-party apps that operate on sensitive user data [11, 21], ensuring they abide by security policies in a mandatory fashion.

Recently, Hails [11], a web-platform framework built atop the LIO IFC system [39, 40], has been used to implement websites that integrate third-party untrusted apps. For example, the code-hosting website `GitStar.com` built with Hails uses untrusted apps to deliver core features, including a code viewer and wiki. GitStar relies on LIO's IFC mechanism to enforce robust privacy policies on user data and code.

LIO, like other IFC systems, ensures that untrusted code does not write data that may have been influenced by sensitive sources to public sinks. For example, an untrusted address-book app is allowed to compute over Alice's friends list and display a stylized version of the list to Alice, but it cannot leak any information about her friends to arbitrary end-points. The flexibility of IFC makes it particularly suitable for the web, where access control lists often prove either too permissive or too restrictive.

However, a key limitation of IFC is the presence of *covert channels*, i.e., "channels" not intended for communication that nevertheless allow code to subvert security policies and share information [22]. A great deal of research has identified and analyzed covert channels [25]. In this work, we focus on the *internal timing covert channel*, which occurs when sensitive data is used to manipulate the timing behavior of threads so that other threads can observe the order in which shared public resources are used [38, 43]. Though we do not believe our solution to the internal timing covert channel affects (either positively or negatively) other timing channels, such as the external timing covert channel, which is derived from measuring external events [1, 5, 12] (e.g., wall-clock), addressing these channels is beyond our present scope.

LIO eliminates the internal timing covert channel by restricting how programmers write code. Programmers are required to explicitly decouple computations that manipulate sensitive data from those that can write to public resources, eliminating covert channels *by construction*. However, decoupling only works when *all* shared resources are modeled. LIO only considers shared resources that are expressible by the programming language, e.g., shared-variables, file descriptors, semaphores, channels, etc. Implicit operating system and hardware state can still be exploited to alter the timing behavior of threads, and thus leak information. Reexamining LIO, we found that the underlying CPU cache

can be used to introduce an internal timing covert channel that leaks sensitive data. A trivial attack can leak data at 0.75 bits/s and, despite the low bandwidth, we were able to leak all the collaborators on a private GitStar.com project in less than a minute.

Several countermeasures to cache-based attacks have previously been considered, primarily in the context of cryptosystems following the work of Kocher [18] (see Section 8). Unfortunately, many of the techniques are not designed for IFC scenarios. For example, modifying an algorithm implementation, as in the case of AES [7], does not naturally generalize to arbitrary untrusted code. Similarly, flushing or disabling the cache when switching protection domains, as suggested in [6, 48], is prohibitively expensive in systems like Hails, where context switches occur hundreds of times per second. Finally, relying on specialized hardware, such as partitioned caches [29], which isolate the effects of one partition from code using a different partition, restricts the deployability and scalability of the solution; partitioned caches are not readily available and often cannot be partitioned to an arbitrary security lattice.

This paper describes a countermeasure for cache-based attacks when execution is confined to a single CPU. Our method generalizes to arbitrary code, imposes minimal performance overhead, scales to an arbitrary security lattice, and leverages hardware features already present in modern CPUs. Specifically, we present an instruction-based scheduler that eliminates internal timing channels in concurrent programs that time-slice a single CPU and contend for the same cache, TLB, bus, and other hardware facilities. We implement the scheduler for the LIO IFC system and demonstrate that, under realistic restrictions, our scheduler eliminates such attacks in Hails web applications.

Our contributions are as follows.

- We implement a cache-based internal timing attack for LIO.
- We close the cache-based covert channel by scheduling user-level threads on a single CPU core based on the number of instructions they execute (as opposed to the amount of time they execute). Our scheduler can be used to implement other concurrent IFC systems which implicitly assume instruction-level scheduling (e.g., [13, 14, 32, 38, 45]).
- We implement our instruction-based scheduler as part of the Glasgow Haskell Compiler (GHC) runtime system, atop which LIO and Hails are built. We use CPU performance counters, prevalent on most modern CPUs, to pre-empt threads according to the number of retired instructions. The measured impact on performance, when compared to time-based scheduling, is negligible.
  We believe these techniques to be applicable to operating systems that enforce IFC, including [20, 26, 46], though at a higher cost in

performance for application code that is highly optimized for locality (see Section 5).

- We augment the LIO [40] semantics to model the cache and formally prove that instruction-based scheduling removes leaks due to caches.

The paper is organized as follows. Section 2 discusses cache-based attacks and existing countermeasures. In Section 3 presents our instruction-based scheduling solution. Section 4 describes our modifications to GHC's runtime, while Section 5 analyses their performance impact. Formal guarantees and discussions of our approach are detailed in Sections 6 and 7. We describe related work in Section 8 and conclude in Section 9.

2 Cache Attacks and Countermeasures

The severity of information leakage attacks through the CPU hardware cache has been widely considered by the cryptographic community (e.g. [28, 31]). Unlike crypto work, where attackers extract sensitive information through the execution of a fixed crypto algorithm, we consider a scenario in which the attacker provides arbitrary code in a concurrent IFC system. In our scenario, the adversary is a developer that implements a Hails app that interfaces with user-sensitive data using LIO libraries.

We found that, knowing only the cache size of the underlying CPU, we can easily build an app that exploits the shared cache to carry out an internal timing attack that leaks sensitive data at 0.75 bits/s. Several IFC systems, including [13, 14, 32, 38, 40, 45], model internal timing attacks and address them by ensuring that the outcome of a race to a public resource does not depend on secret data. Unfortunately, these systems only account for resources explicitly modeled at the programming language level and not underlying OS or hardware state, such as the CPU cache or TLB. Hence, even though the semantics of these systems rely on instruction-based scheduling (usually to simplify expressing reduction rules), real-world implementations use time-based scheduling for which the formal guarantees do not hold. The instruction-based scheduler proposed in this work can be used to make the assumptions of such concurrent IFC systems match the situation in practice. In the remainder of this section, we show the internal timing attack that leverages the hardware cache. We also discuss several existing countermeasures that could be employed by Hails.

2.1 Example cache attack

We mount an internal timing attack by influencing the scheduling behavior of threads through the cache. Consider the code shown in Figure 1. The attack leaks the secret boolean value `secret` in thread 1 by affecting when thread 2 writes to the public channel relative to thread 3.

```
1.                                  lowArray := new Array[M];
2.                                  fillArray(lowArray)
```

thread 1	thread 2	thread 3
1. if secret	1. for i in [1..n]	1. for i in [1..n+m]
2. then highArray := new Array[M]	2. skip	2. skip
3. fillArray(highArray)	3. readArray(lowArray)	3. outputLow(0)
4. else skip	4. outputLow(1)	

Fig. 1. A simple cache attack.

The program starts (lines 1–2) by creating and initializing a public array `lowArray` whose size `M` corresponds to the cache size; `fillArray` simply sets every element of the array to `0` (this will place the array in the cache). The program then spawns three threads that run concurrently. Assuming a round-robin time-based scheduler, the execution of the attack proceeds as illustrated in Figure 2, where `secret` is set to true (top) and false (bottom), respectively.

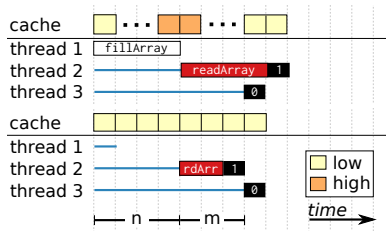

Fig. 2. Execution of the cache attack with `secret` true (top) and false (bottom).

- Depending on the secret value `secret`, thread 1 either performs a no-operation (`skip` on line 4), leaving the cache intact, or evicts `lowArray` from the cache (lines 2–3) by creating and initializing a new (non-public) array `highArray`.
- We assume that thread 1 takes less than `n` steps to complete its execution—a number that can be determined experimentally; in Figure 2, `n` is four. Hence, to allow all the effects on the cache due to thread 1 to settle, thread 2 delays its computation by `n` steps (lines 1–2). Subsequently, the thread reads every element of the public array `lowArray` (line 3), and finally writes `1` to a public output channel (line 4). Crucial to carrying out the attack, the duration of thread 2's reads (line 3) depends on the state of the cache: if the cache was modified by thread 1, i.e., `secret` is true, thread 2 needs to wait for all the public data to be retrieved from memory (as opposed to the cache) before producing an output. This requires evicting `highArray` from the cache and fetching `lowArray`, a process that takes a non-negligible amount of time. However, if the cache was not touched by thread 1, i.e., `secret` is false, thread 2 will get few cache misses and thus produce its output with no delay.
- We assume that thread 2 takes less than `m`, where `m`$<$`n`, steps to complete reading `lowArray` (line 3) when the reads hit the cache, i.e., `lowArray` was not replaced by `highArray`. Like `n`, this metric can be determined experimentally; in Figure 2, `m` is three. Using this, thread

thread 3 simply delays its computation by `n+m` steps (lines 1–2) and then writes `0` to a public output channel (line 3). The role of thread 3 is solely to serve as a baseline for thread 2's output: producing its output before thread 2 when the latter is filling the cache, i.e., `secret` is true; conversely, it produces an output after thread 2 if thread 1 did not touch the cache, i.e., `secret` is false.

We remark that the race between thread 2 and thread 3 to write to a shared public channel, influenced by the cache state, is precisely what facilitates the attack. We described how to leak a single bit, but the attack can easily be magnified by wrapping it in a loop. Note also that we have assumed the attacker has complete control of the cache—i.e., the cache is not affected by other code running in parallel. However, the attack is still plausible under weaker assumptions so long as the attacker deals with the additional noise, as exemplified by the timing attacks on AES [28].

2.2 Existing countermeasures

The internal timing attack arises as a result of cache effects influencing thread-scheduling behavior. Hence, one series of countermeasures addresses the problem through low-level CPU features that provide better control of the cache.

Flushing the cache Naively, we can flush the cache on every context switch. In the context of Figure 1, this guarantees that, when thread 2 executes the `readArray` instruction, its duration is not affected by thread 1 evicting `lowArray` from the cache—the cache will *always* be flushed on a context switch, hence thread 3 will always write to the output channel first.

No-fill cache mode Several architectures, including Intel's Xeon and Pentium 4, support a cache *no-fill* mode [15]. In this mode, read/write hits access the cache; misses, however, read from and write to memory directly, leaving the cache unchanged. As considered by Zhang et al. [48], we can execute all threads that operate on non-public data in this mode. This approach guarantees that sensitive data cannot affect the cache. Unfortunately, threads operating on non-public data and relying on the cache will suffer from performance degradation.

Partitioned cache Another approach is to partition the cache according to the number of security levels, as suggested in [48]. Using this architecture, a thread computing on secret data only accesses the secret partition, while a thread computing on public data only access the public one. This approach effectively corresponds to giving each differently-labeled thread access to its own cache and, as a result, the scheduling behavior of public threads cannot be affected by evicting data from the cache.

Unfortunately, none of the aforementioned solutions can be used in systems built with Hails (e.g., GitStar). Flushing the cache is prohibitively expensive for preemptive systems that perform a context switch hundreds of times per second—the impact on performance would gravely reduce usability. The no-fill mode solution is well suited for systems wherein the majority of the threads operate on public data. In such cases, only threads operating on sensitive data will incur a performance penalty. However, in the context of Hails, the solution is only slightly less expensive than flushing the cache. Hails threads handle HTTP requests that operate on individual (non-public) user data, hence most threads will not be using the cache. Another consequence of threads handling differently-labeled data is that partitioned caches can only be used in a limited way (see Section 8). Specifically, to address internal timing attacks, it is required that we partition the cache according to the number of security levels in the lattice. Given that most existing approaches can only partition caches up to 16-ways at the OS level [24], and fewer at the hardware level, an alternative scalable approach is necessary. Moreover, neither flushing nor partitioning the cache can handle timing perturbations arising from other pieces of hardware such as the TLB, buses, etc.

3 Instruction-based Scheduling

As the example in Figure 2 shows, races to acquire public resources are affected by the cache state, which in turn might be affected by secret values. It is important to highlight that the number of instructions executed in a given quantum of time might vary depending on the state of the cache. It is precisely this variability that reintroduces dangerous races into systems. However, the actual set of instructions executed is not affected by the cache. Hence, we propose scheduling threads according to the number of instructions they execute, rather than the amount of time they consume. The point at which a thread produces an output (or any other visible operation) is determined according to the number of instructions it has executed, a measurement unaffected by the amount of time it takes to perform a read/write from memory.

Consider the code in Figure 1 executing atop an instruction-based scheduler. An illustration of this is shown in Figure 3. For simplicity of exposition, the instruction granularity is at the level of commands (`skip`, `readArray`, etc.) and therefore context switches are triggered after one command gets executed. (In Section 4, we describe a more practical and realistic instruction-based scheduler.) Observe that the amount of time it takes to execute an instruction has not changed from the time-based scheduler of Figure 2. For example, `readArray` still takes 6 units of time when `secret` is true, and 2 when it is false. Unlike Figure 2, however, the interleaving between thread 2 and thread 3 did not change depend-

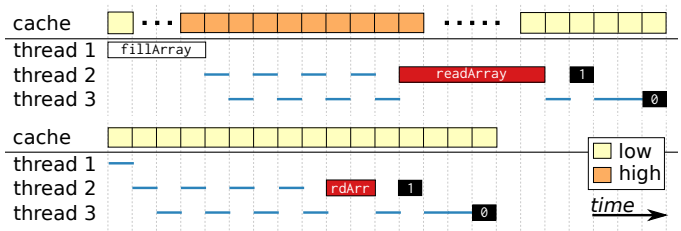

Fig. 3. Execution of cache attack program of Figure 1 with `secret` set to true (top) and false (bottom). In both executions, we highlight that the threads execute one "instruction" at a time in a round-robin fashion. The concurrent threads take the same amount of time to complete execution as in Figure 2. However, since we use instructions to context switch threads, the interleaving between thread 2 or 3 is not influenced by the actions in thread 1, and thus the internal timing attack does not arise—the threads' output order cannot encode sensitive data.

ing on the state of the cache (which did change according to `secret`). Therefore, a race to write to the public channel between thread 2 and thread 3 cannot be caused by the secret, through the cache. The second thread always executes $\texttt{n}+1 = 5$ instructions before writing `1` to the public channel, while the third thread always executes $\texttt{n}+\texttt{m}+1 = 8$ instructions before writing `0`.

Our proposed countermeasure, the implementation of which is detailed in Section 4, eliminates the cache-based internal timing attacks without sacrificing scalability and with a minor performance impact. With instruction-based scheduling, we do not require flushing of the cache. In this manner, applications can safely utilize the cache to retain most of their performance without giving up system security, and unlike current partitioned caches, we can scale up to consider arbitrarily complex lattices.

4 Implementation

We implemented an instruction-based scheduler for LIO. In this section, we describe this implementation and detail some key design features we believe to be useful when modifying concurrent IFC systems to address cache-based timing attacks.

4.1 LIO and Haskell

LIO is a Haskell library that exposes concurrency to programmers in the form of "green," lightweight threads. Each LIO thread is a *native* Haskell thread that has an associated security level (label) which is used to track and control the flow of information to/from the thread. LIO relies on

Haskell libraries for creating new threads and the runtime system for managing them.

In general, M lightweight Haskell threads may concurrently execute on N OS threads. (It is common, however, for multiple Haskell threads to execute on a single OS thread, i.e., $M : 1$ mapping.) The Haskell runtime, as implemented by the GHC system, uses a round-robin scheduler to context switch between concurrently executing threads. Specifically, the scheduler is invoked whenever a thread blocks/terminates or a timer signal alarm is received. The timer is used to guarantee that the scheduler is periodically executed, allowing the runtime to implement preemptive scheduling.

4.2 Instruction-based scheduler

As previously mentioned, timing-based schedulers render systems, such as LIO, vulnerable to cache-based internal timing attacks. We implement our instruction-based scheduler as a drop-in replacement for the existing GHC scheduler, using the number of retired instructions to trigger a context switch.

Specifically, we use performance monitoring units (PMUs) present in almost all recent Intel [15] and AMD [3] CPUs. PMUs expose hardware performance counters that are typically used by developers to optimize code—they provide metrics such as the number of cache misses, instructions executed per cycle, branch mispredictions, etc. Importantly, PMUs also provide a means for counting the number of retired instructions.

Using the perfmon2 [9] Linux monitoring interface and helper user-level library `libpfm4`, we modified the GHC runtime to configure the underlying PMU to count the number of retired instructions the Haskell process is executing. Specifically, with perfmon2 we set a data performance counter register to $2^{64} + n$, which the CPU increments upon retiring an instruction.[1] Once the counter overflows, i.e., n instructions have been retired, perfmon2 is sent a hardware interrupt. In our implementation, we configured perfmon2 to handle the interrupt by delivering a signal to the GHC runtime.

If threads share no resources, upon receiving a signal, the executing Haskell thread can immediately save its state and jump to the scheduler. However, preempting a thread which is operating on a shared memory space can be dangerous, as the thread may have left memory in an inconsistent state. (This is the case for many language runtimes, not solely GHC's.) To avoid this, GHC produces code that contains *safe points* where threads may yield. Hence, a signal does not cause an immediate preemption. Instead, the signal handler simply sets a flag indicating the arrival

[1] Though the bit-width of the hardware counters vary (they are typically 40-bits wide) perfmon2 internally manages a 64-bit counter.

of a signal; at the next safe point, the thread "cooperatively" yields to the scheduler.

To ensure liveness, we must guarantee that given any point in execution, a safe point is reached in n instructions. Though GHC already inserts many safe points as a means of invoking the garbage collector (via the scheduler), tight loops that do not perform any allocation are known to hang execution [10]. Addressing this eight-year old bug, which would otherwise be a security concern in LIO, we modified the compiler to insert safe points on function entry points. This modification, integrated in the mainline GHC, has almost no effect on performance and only a 7% bloat in average binary size.

4.3 Handling IO

Threads yield at safe points in their execution paths as a result of a retired instruction signal. However, there are circumstances in which threads would like to explicitly yield prior to the reception of a retired instruction signal. In particular, when a thread performs a blocking operation, it immediately yields to the scheduler, registering itself to wake up when the operation completes. Thus, any IO action is a yield which allows the thread to give up the rest of its scheduling quantum.

While yields are not intrinsically unsafe, it is not safe to allow the leftover scheduling quantum to be passed on to the next thread. Thus, after running any asynchronous IO action, the runtime must reset the retired instruction counter. Hence, whenever a thread enters the scheduler loop due to being blocked, we reset the retired instruction counter.

5 Performance Evaluation

We evaluated the performance of instruction-based scheduling against existing time-based approaches using the nofib benchmark suite [30]. nofib is the standard benchmarking suite used for measuring the performance of Haskell implementations.

In our experimental setup, we used the latest development version of GHC (the Git master branch as of November 6, 2012). The measurements were taken on the same hardware as Hails [11]: a machine with two dual-core Intel Xeon E5620 (2.4GHz) processors, and 48GB of RAM.

We first needed to find an instruction budget—number of instructions to retire before triggering the scheduler. We found a poorly chosen instruction budget could increase runtime by 100%. To determine a good parameter, we measured the mean time between retired-instruction signals with an initially guessed instruction budget parameter. We then adjusted the parameter so the median test program had a 10 millisecond mean time-slice (the default quantum size in vanilla GHC with

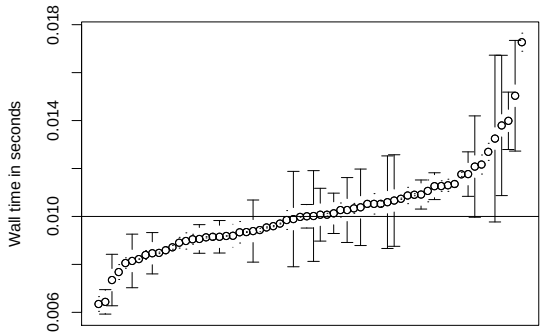

Fig. 4. Mean time between timer signal and retired-instruction signal. Each point represents a program from nofib, which have been sorted on the x-axis by their mean time.

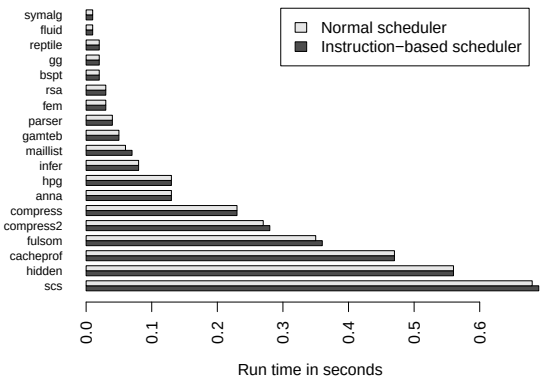

Fig. 5. Change to run time from instruction-based scheduling

time-based scheduling) and verified our final choice by re-running the measurements. For our specific setup, an instruction budget of approximately 37,100,000 retired-instructions corresponded to a 10 millisecond time quantum. We plot the mean and standard deviation across all nofib applications with the final tuning parameter in Figure 4. We found that most programs receive a signal within 2 milliseconds of when they would have normally received the signal using the standard time-based scheduler. While the instruction budget parameter will vary across machines, it is relatively simple to bootstrap this parameter by performing these measurements at startup and tuning the budget accordingly.

Next, we compared the performance of Haskell's timer-based scheduler with our instruction-based scheduler. We used a subset of the nofib benchmark suite called the real benchmark, which consists of "real world

programs", as opposed to synthetic benchmarks (however, results for the whole nofib suite are comparable). Figure 5 shows the run time of these programs with both scheduling approaches. With an optimized instruction budget parameter, instruction-based scheduling has no impact to the runtime of the majority of nofib applications and results in only a very slight increase in runtime for others (about 1%).

This result may seem surprising: instruction-based scheduling purposely punishes threads with good data locality, so one might expect a more substantial performance impact. We hypothesize that this is the case due to two reasons. First, with preemptive scheduling, we are already inducing cache misses when we switch from running one thread to another—instruction-based scheduling only perturbs when these preempts occur, and as seen in Figure 4, these perturbations are very minor. Second, modern L2 caches are quite large, meaning that hardware is more forgiving of poor data locality—an effect that has been measured in the behavior of stock lazy functional programs [2].

6 Cache-aware semantics

In this section we recall relevant design aspects of LIO [40] and extend the original formalization to consider how caches affect the timing behavior of programs. Importantly, we formalize instruction-based scheduling and show how it removes cache-based internal timing covert channels.

6.1 LIO Overview

At a high level, LIO provides the `LIO` monad, which is used in place of `IO`. Wrapping standard Haskell libraries, LIO exports a collection of functions that untrusted code may use to access the filesystem, network, shared variables, etc. Unlike the standard libraries, which usually return `IO` actions, these functions return actions in the `LIO` monad, thus allowing LIO to perform label checks before executing a potentially unsafe action.

Internally, the `LIO` monad keeps track of a *current label*, L_{cur}. The current label is effectively a ceiling over the labels of all data that the current computation may depend on. This label eliminates the need to label individual definitions and bindings: symbols in scope are (conceptually) labeled with L_{cur}.[2] Hence, when a computation C, with current label L_C, observes an object labeled L_O, C's label is raised to the least upper bound or *join* of the two labels, written $L_C \sqcup L_O$. Importantly the current label governs where the current computation can write, what labels may

[2] As described in [39], LIO does, however, allow programmers to heterogeneously label data they consider sensitive.

be used when creating new channels or threads, etc. For example, after reading O, the computation should not be able to write to a channel K if L_C is more restricting than L_K—this would potentially leak sensitive information (about O) into a less sensitive channel.

Note that an LIO computation can only execute a sub-computation on sensitive data by either raising its current label or forking a new thread in which to execute this sub-computation. In the former case, raising the current label prevents writing to less sensitive endpoints. In the latter case, to observe the result (or timing and termination behavior) of the sub-computation the thread must wait for the forked thread to finish, which first raises the current label. A consequence of this design is that differently-labeled computations are decoupled, which, as mentioned in Section 1, is key to eliminating the internal timing covert channel.

In the next subsection, we will outline the semantics for a cache-aware, time-based scheduler where the cache attack described in Section 2 is possible. Moreover, we show that we can easily adapt this semantics to model the new LIO instruction-based scheduler.

6.2 Cache-aware semantics

We model the underlying CPU cache as an abstract memory shared among all running threads, which we will denote with the symbol ζ. Every step of the sequential execution relation will affect ζ according to the current instruction being executed, the runtime environment, and the existing state of the cache. As in [40], each LIO thread has a thread-local runtime environment σ, which contains the current label $\sigma.\texttt{lbl}$. The global environment Σ, common to all threads, holds references to shared resources.

In addition, we explicitly model the number of machine cycles taken by a single execution step as a result of the cache. Specifically, the transition $\zeta \rightarrow_k^{(\Sigma,\sigma,e)} \zeta'$ captures the parameters that influence the cache (Σ, σ, and e) as well as the number of cycles k it takes for the cache to be updated.

A *cache-aware* evaluation step is obtained by merging the reduction rule of LIO with our formalization of CPU cache as given below:

$$\frac{\langle \Sigma, \langle \sigma, e\rangle\rangle \xrightarrow{\gamma} \langle \Sigma', \langle \sigma', e'\rangle\rangle \qquad \zeta \rightarrow_k^{(\Sigma,\sigma,e)} \zeta' \qquad k \geq 1}{\langle \Sigma, \langle \sigma, e\rangle\rangle_\zeta \xrightarrow{\gamma}_k \langle \Sigma', \langle \sigma', e'\rangle\rangle_{\zeta'}}$$

We read $\langle \Sigma, \langle \sigma, e\rangle\rangle_\zeta \xrightarrow{\gamma}_k \langle \Sigma', \langle \sigma', e'\rangle\rangle_{\zeta'}$ as "the configuration $\langle \Sigma, \langle \sigma, e\rangle\rangle$ reduces to
$\langle \Sigma', \langle \sigma', e'\rangle\rangle$ in one step, but k machine cycles, producing event γ and modifying the cache from ζ to ζ'." As in LIO [40], the relation $\langle \Sigma, \langle \sigma, e\rangle\rangle \xrightarrow{\gamma}$

$$\frac{\langle \Sigma, \langle \sigma, e\rangle\rangle_\zeta \longrightarrow_k \langle \Sigma', \langle \sigma', e'\rangle\rangle_{\zeta'} \qquad q > 0}{\langle \Sigma, \zeta, q, \langle \sigma, e\rangle \lhd t_s\rangle \hookrightarrow \langle \Sigma', \zeta', q + k, \langle \sigma', e'\rangle \lhd t_s\rangle} \text{(STEP)}$$

$$\frac{q \leq 0}{\langle \Sigma, \zeta, q, t \lhd t_s\rangle \hookrightarrow \langle \Sigma', \zeta, q_i, t_s \rhd t\rangle} \text{(PREEMPT)}$$

Fig. 6. Semantics for threadpools under round-robin time-based scheduling

$\langle \Sigma', \langle \sigma', e'\rangle\rangle$ represents a single execution step from thread expression e, under the run-time environments Σ and σ, to thread expression e' and run-time environments Σ' and σ'. Events are used to communicate information between the threads and the scheduler, e.g., when spawning new threads.

Figure 6 shows the most important rules of our time-based scheduler in the presence of cache effects. We elide the rest of the rules for brevity. The relation $\hookrightarrow$ represents a single evaluation step for the program threadpool, in contrast with $\longrightarrow$ which is only for a single thread. Configurations are of the form $\langle \Sigma, \zeta, q, t_s\rangle$, where q is the number of cycles available in the current time slice and t_s is a queue of thread configurations of the form $\langle \sigma, e\rangle$. We use a standard deque-like interface with operations $\lhd$ and $\rhd$ for front and back insertion, respectively, i.e., $\langle \sigma, e\rangle \lhd t_s$ denotes a threadpool in which the first thread is $\langle \sigma, e\rangle$ while $t_s \rhd \langle \sigma, e\rangle$ indicates that $\langle \sigma, e\rangle$ is the last one.

As in LIO, threads are scheduled in a round-robin fashion. Our scheduler relies on the number of cycles that each step takes; we respectively write q_i and q as the initial and remaining number of cycles assigned to a thread in each quantum. In rule (STEP), the number of cycles k that the current instruction takes is reflected in the scheduling quantum. Consequently, threads that compute on data that is not present in the cache will take more cycles, i.e., have a higher k, so they will run "slower" because they are allowed to perform fewer reduction steps in the remaining time slice. In practice, this permits attacks, such as that in Figure 1, where the interleaving of the threads can be affected by sensitive data. Rule (PREEMPT) is used when the thread has exhausted its cycle budget, triggering a context switch by moving the current thread to the end of the queue.

We can adapt this semantics to reflect the behavior of the new instruction-based scheduler. To this end, we replace the number of cycles q with an instruction budget; we write b_i for the initial instruction budget and b for the current budget. Crucially, we change rule (STEP) into rule (STEP-CA), given by

$$\text{(STEP-CA)}\quad \frac{\langle \Sigma, \langle \sigma, e\rangle\rangle_\zeta \longrightarrow_k \langle \Sigma', \langle \sigma', e'\rangle\rangle_{\zeta'} \qquad b > 0}{\langle \Sigma, \zeta, b, \langle \sigma, e\rangle \triangleleft t_s\rangle \hookrightarrow \langle \Sigma', \zeta', b+1, \langle \sigma', e'\rangle \triangleleft t_s\rangle}.$$

Rule (STEP-CA) executes a sequential instruction in the current thread, provided the instruction budget is not empty ($b > 0$), and updates the cache accordingly ($\langle \Sigma, \langle \sigma, e\rangle\rangle_\zeta \longrightarrow_k \langle \Sigma', \langle \sigma', e'\rangle\rangle_{\zeta'}$). It is important to remark that the effects of the underlying cache ζ, as indicated by k, are intentionally ignored by the scheduler. This subtle detail captures the essence of removing the cache-based internal timing channel. (Our formalization of a time-based scheduler does not ignore k and thus is vulnerable.) Similarly, rule (PREEMPT) turns into rule (PREEMPT-CA), where q and q_i are respectively replaced with b and b_i to reflect the fact that there is an instruction budget instead of a cycle count. The rest of the rules can be adapted in a straightforward manner. Our rules have the invariant that the instruction budget gets decremented by one when a thread executes one instruction.

By changing the cache-aware semantics in this way, we obtain a generalized semantics for LIO. In fact, the previous semantics for LIO [40], is a special case, with $b_i = 1$, i.e., the threads perform only one reduction step before a context-switch happens. In addition, it is easy to extend our previous termination-sensitive non-interference result to the instruction-based semantics. The security guarantees of our approach are stated below.

Theorem 1 (Termination-sensitive non-interference). *Given a program function f, an attacker that observes data at level L, and a pair of inputs e_1 and e_2 indistinguishable to the attacker, then for every reduction sequence starting from $f(e_1)$ there is a corresponding reduction sequence starting from $f(e_2)$ such that both sequences reach indistinguishable configurations.*

Proof Sketch: Our proof relies on the *term erasure technique* as used in [23, 34, 39], and follows in a similar fashion to that of [40]. More details can be found in Appendix 1.

7 Limitations

This section discusses some limitations of our current implementation, the significance of these limitations, and how the limitations can be addressed.

Nondeterminism in the hardware counters While the retired-instruction counter should be deterministic, in most hardware implementations there is some degree of nondeterminism. For example, on most x86 processors

the instruction counter adds an extra instruction every time a hardware interrupt occurs [44]. This anomaly could be exploited to affect the behavior of an instruction-based scheduler, causing it to trigger a signal early. However, this is only a problem if a high thread is able to cause a large number of hardware interrupts in the underlying operating system. In the Hails framework, attackers can trigger interrupts by forcing a server to frequently receive HTTP responses, i.e., trigger a hardware interrupt from the network interface card. Hails, however, provides mechanisms to mitigate the effects of external events, using the techniques of [4, 47], that can reduce the frequency of such operations. Nevertheless, the feasibility of such attacks is not directly clear and left as future work.

Scheduler and garbage collector instruction counts For performance reasons, we do not reset the retired-instruction counter prior to re-entering user code. This means that instruction counts include the instructions executed from when the previous thread received the signal, to when the previous thread yields, to when the next thread is scheduled. While this suggests that thread are not completely isolated, we think that this interaction is extremely difficult to exploit. This is because the number of instructions it takes for the scheduler to schedule a new thread is essentially fixed, and the "time to yield" for any code is highly dependent on the compiler, which we assume is not under the control of an adversary.

Parallelism Unfortunately, we cannot simply run instruction-based scheduling on multiple cores. Threads running in parallel will be able to race to public resources. Under normal conditions, such races can be still influenced by the state of the (L3) cache. Some parallelism is, however, possible. For instance, we can extend the instruction-based scheduler to parallelize regions of code that do not share state or have side effects (e.g., synchronization operations or writes to channels). To this end, when a thread wishes to perform a side effect, it is required that all the other threads lagging behind (as per retired-instruction count) first complete the execution of their side effects. Hence, an implementation would rely on a synchronization barrier whenever a side-effecting computation is executed; at the barrier, the execution of all the side effects is done in a pre-determined order. Although we believe that this "optimization" is viable, we have not implemented it, since it requires major modifications to the GHC runtime system and the performance gains due to parallelism requiring such strict synchronization barriers are not clear. We leave this investigation to future work.

Even without built-in parallelism, we believe that instruction-based scheduling represents a viable and deployable solution when considering modern web applications and data-centers. In particular, when an application is distributed over multiple machines, these machines do not

share a processor cache and thus can safely run the application concurrently. Attacks which involve making these two machines access shared external resources can be mitigated in the same fashion as external timing attacks [4, 40, 47, 48]. Load-balancing an application in this manner is already a well-established technique for deploying applications.

8 Related work

Impact of cache on cryptosystems Kocher [18] was one of the first to consider the security implications of memory access-time in implementations of cryptographic primitives and systems. Since then, several attacks (e.g., [28, 31]) against popular systems have successfully extracted secret keys by using the cache as a covert channel. As a countermeasure, several authors propose partitioning the cache (e.g., [29]). Until recently, partitioned caches have been of limited application in dynamic information flow control systems due to the small number of partitions available. The recent Vantage cache partition scheme of Sanchez and Kozyrakis [37], however, offers tens to hundreds of configurable partitions and high performance. As hardware is not yet available with Vantage, it is hard to evaluate its effectiveness for our problem domain. However, we expect it to be mostly complimentary to our instruction-based scheduler. Specifically, a partitioned cache can be used to safely run threads in parallel, each group of threads using instruction-based schedulers. Other countermeasures (e.g., [28]) are primarily implementation-specific, and, while applicable to cryptographic primitives, they do not easily generalize to arbitrary code.

Language-based information-flow security Several works (e.g., [13]) consider systems that satisfy *possibilistic non-interference* [38], which states that a concurrent program is secure iff the possible observable events do not depend on sensitive data. An alternative notion, *probabilistic non-interference*, considers a concurrent program secure iff the probability distribution over observable events is not affected by sensitive data [43]. Zdancewic and Myers introduce *observational low-determinism* [45], which intuitively states that the observable behavior of concurrent systems must be deterministic. After this seminal work, several authors improve on each other's definitions on low-determinism (e.g., [14]). Other IFC systems rely on deterministic semantics and a determined class of runtime schedulers (e.g., [32]).

The lines of work mentioned above assume that the execution of a single step is performed in a single unit of time, corresponding to an instruction, and show that races to publicly-observable events cannot be influenced by secret data. Unfortunately, the presence of the cache breaks the correspondence between an instruction and a single unit of

time, making cache attacks viable. Instruction-based scheduling could be seen as a necessary component in making the previous concurrent IFC approaches practical.

Agat [1] presents a code transformation for sequential programs such that both code paths of a branch have the same memory access pattern. This eliminates timing covert channels, even those relying on the cache. This transformation has been adapted by several authors (e.g., [36]). This approach, however, focuses on avoiding attacks relying on the data cache, while leaving the instruction cache unattended.

Russo and Sabelfeld [33] consider non-interference for concurrent systems under cooperative and deterministic scheduling. An implementation of such a system was presented by Tsai et al. in [41]. This approach eliminates internal timing leaks, including those relying on the cache, by restricting the use of yields. Cooperative schedulers are intrinsically vulnerable to attacks that use termination as a covert channel. In contrast, our solution is able to safely preempt non-terminating computations while guaranteeing termination-sensitive non-interference.

Secure multi-execution [8] preserves confidentiality of data by executing the same sequential program several times, one for each security level. In this scenario, the cache-based covert channel can only be removed in specific configurations [16]. Zhang et al. [48] provide a method to mitigate external events when their timing behavior could be affected by the underlying hardware. This solution is directly applicable to our system when considering external events. Similar to our work, they consider an abstract model of the hardware machine state which includes a description of time. However, their semantics focus on sequential programs, wherein attacks due to the cache arise in the form of externally visible events.

Hedin and Sands [12] present a type-system for preventing external timing attacks for bytecode. Their semantics is augmented to incorporate history, which enables the modeling of cache effects. We proceed in a similar manner when extending the original LIO semantics [40] to consider caches.

System security In order to achieve strong isolation, Barthe et al. [6] present a model of virtualization which flushes the cache upon switching between guest operating systems. Different from our scenario, flushing the cache in such scenarios is common and does not impact the already-costly context-switch.

Allowing some information leakage, Köpft et al. [19] combines abstract interpretation and quantitative information-flow to analyze leakage bounds for cache attacks. Kim et al. [17] propose StealthMem, a system level protection against cache attacks. StealthMem allows programs to allocate memory which does not get evicted from the cache. In fact, this approach could be seen as a software-level partition of the

cache. StealthMem is capable of enforcing confidentiality for a stronger attacker model than ours, i.e., they consider programs with access to wall-clock and perhaps running on multi-cores. As other works on partition caches, StealthMem does not scale to scenarios with arbitrarily complex security lattices.

Performance monitoring counters The use of PMUs for tasks other than performance monitoring is a relatively recent one. Vogl and Ekert [42] also use PMUs, but for monitoring applications running within a virtual machine, allowing instruction level monitoring of all or specific instructions. While the mechanism is the same, our goals are different: we merely seek to replace interrupts generated by a clock-based timer with interrupts generated by hardware counters; their work introduces new interrupts that trigger vmexits. This causes a considerable slowdown, while we achieve no major performance impact.

9 Conclusion

Cache-based internal timing attacks constitute a practical set of attacks. We present instruction-based scheduling as a solution to remove such attacks. Different from simply flushing the cache on a context switch or partitioning the cache, this new class of schedulers also removes timing perturbations introduced by other components of the underlying hardware (e.g., the TLB, CPU buses, etc.). To demonstrate the applicability of our solution, we implemented a scheduler using the CPU retired-instruction counters available on commodity Intel and AMD hardware. We integrated the scheduler into the Hails IFC web framework, replacing the timing-based scheduler. This integration was, in part, possible because of the scheduler's negligible performance impact and, in part, due to our formal guarantees. Specifically, by generalizing previous results, we proved that instruction-based scheduling for LIO preserves confidentiality and integrity of data, i.e., termination-sensitive non-interference. Finally, we remark that our design, implementation, and proof are not limited to LIO; we believe that instruction-based scheduling is applicable to other concurrent deterministic IFC systems where cache-based timing attacks could be a concern.

Acknowledgments

This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by multiple gifts from Google, and by the Swedish research agency VR and STINT. Deian Stefan is supported by the DoD through the NDSEG Fellowship Program.

References

[1] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, Jan. 2000.

[2] A. Ahmad and H. DeYoung. Cache performance of lazy functional programs on current hardware. Technical report, CMU, December 2009.

[3] AMD. BIOS and kernel developer's guide for AMD family 11h processors, July 2008.

[4] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proc. of the 17th ACM CCS*. ACM, 2010.

[5] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. *Electron. Notes Theor. Comput. Sci.*, 153, May 2006.

[6] G. Barthe, G. Betarte, J. Campo, and C. Luna. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE Computer Society, june 2012.

[7] J. Bonneau and I. Mironov. Cache-collision timing attacks against AES. *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 201–215, 2006.

[8] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10. IEEE Computer Society, 2010.

[9] S. Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. of the 2006 Ottawa Linux Symposium*, pages 269–288. Citeseer, 2006.

[10] GHC. Infinite loops can hang Concurrent Haskell. `http://hackage.haskell.org/trac/ghc/ticket/367`, 2005.

[11] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. of the 10th Symposium on Operating Systems Design and Implementation*, October 2012.

[12] D. Hedin and D. Sands. Timing aware information flow security for a JavaCard-like bytecode. *Elec. Notes Theor. Comput. Sci.*, 141, 2005.

[13] K. Honda, V. T. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. of the 9th European Symposium on Programming Languages and Systems*. Springer-Verlag, 2000.

[14] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. IEEE Computer Sec. Foundations Workshop*, July 2006.

[15] Intel. Intel 64 and IA-32 architectures software developer's manual, August 2012.

[16] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. of IEEE Symposium on Sec. and Privacy*. IEEE, 2011.

[17] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12. USENIX Association, 2012.

[18] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. of the 16th CRYPTO*. Springer-Verlag, 1996.

[19] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In *Proc. of the Intl. conference on Computer Aided Verification*. Springer-Verlag, 2012.

[20] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st Symp. on Operating Systems Principles*, October 2007.

[21] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A World Wide Web Without Walls. In *6th ACM Workshop on Hot Topics in Networking (Hotnets)*, Nov. 2007.

[22] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[23] P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.

[24] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proc. of the Intl. Symposium on High Performance Computer Architecture*. IEEE, 2008.

[25] J. Millen. 20 years of covert channel modeling and analysis. In *IEEE Symp. on Security and Privacy*, 1999.

[26] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: from general purpose to a proof of information flow enforcement. In *Proceedings of the34th IEEE Symp. on Security and Privacy*, 2013.

[27] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 129–142, 1997.

[28] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology*, CT-RSA'06. Springer-Verlag, 2006.

[29] D. Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptology ePrint Archive*, 2005, 2005.

[30] W. Partain. The nofib benchmark suite of Haskell programs. *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, 1992.

[31] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[32] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 177–189, July 2006.

[33] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics (PSI)*, LNCS. Springer-Verlag, June 2006.

[34] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN Symposium on Haskell*, pages 13–24. ACM Press, Sept. 2008.

[35] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[36] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 200–214, July 2000.

[37] D. Sanchez and C. Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *International Symposium on Computer Architecture*. ACM IEEE, 2011.

[38] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[39] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

[40] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the 17th ACM SIGPLAN International Conference on Functional Programming*, Sep. 2012.

[41] T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. IEEE Computer Sec. Foundations Symposium*, July 2007.

[42] S. Vogl and C. Eckert. Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture. *Proceedings of the 2012 European Workshop on System Security EuroSec'12*, 2012.

[43] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3), Nov. 1999.

[44] V. M. Weaver and S. A. McKee. Can hardware performance counters be trusted? *Workload Characterization.*, 08, 2008. `http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=4636099`.

[45] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 29–43, June 2003.

[46] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

[47] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proc. of the 18th ACM CCS*. ACM, 2011.

[48] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proc. of PLDI*. ACM, 2012.

1 Formalization of LIO with instruction-based scheduling

LIO is formalized as a simply typed Curry-style call-by-name λ-calculus with some extensions. Figure 7 defines the formal syntax for the language. Syntactic categories v, e, and τ represent values, expressions, and types, respectively.

The values in the calculus have their usual meaning for typed λ-calculi. Symbol m represents `LMVars`. Special syntax nodes are added to this category: `Lb` v e, $(e)^{\texttt{LIO}}$, `R` m, and $\boxdot$. Node `Lb` v e denotes the run-time representation of a labeled value. Similarly, node $(e)^{\texttt{LIO}}$ denotes the run-time result of a monadic `LIO` computation. Node $\boxdot$ denotes the run-time representation of an empty `LMVar`. Node `R` m is the run-time representation of a `Result`, implemented as a `LMVar`, that is used to access the result produced by spawned computations.

Label:	l
LMVar:	m
Value:	$v ::= \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid l \mid m \mid x \mid \lambda x.e$
	$\mid \texttt{fix}\ e \mid \texttt{Lb}\ l\ e \mid (e)^{\texttt{LIO}} \mid \boxdot \mid \texttt{R}\ m$
Expression:	$e ::= v \mid \bullet \mid e\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
	$\mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{return}\ e \mid e \gg\!= e$
	$\mid \texttt{label}\ e\ e \mid \texttt{unlabel}\ e \mid \texttt{getLabel}$
	$\mid \texttt{labelOf}\ e \mid \texttt{lFork}\ e\ e \mid \texttt{lWait}\ e$
	$\mid \texttt{newLMVar}\ e\ e \mid \texttt{takeLMVar}\ e$
	$\mid \texttt{putLMVar}\ e\ e \mid \texttt{labelOfLMVar}\ e$
Type:	$\tau ::= \texttt{Bool} \mid \texttt{()} \mid \tau \to \tau \mid \ell \mid \texttt{Labeled}\ \ell\ \tau$
	$\mid \texttt{Result}\ \ell\ \tau \mid \texttt{LMVar}\ \ell\ \tau \mid \texttt{LIO}\ \ell\ \tau$

Fig. 7. Syntax for values, expressions, and types.

Expressions are composed of values (v), the special node $\bullet$, representing an erased term, function applications ($e\ e$), conditional branches (`if` e `then` e `else` e), and local definitions (`let` $x = e$ `in` e). Additionally, expressions may involve operations related to monadic computations in the `LIO` monad. More precisely, `return` e and $e \gg\!= e$ represent the monadic return and bind operations. Monadic operations related to the manipulation of labeled values inside the `LIO` monad are given by `label` and `unlabel`. Expression `unlabel` e acquires the content of the labeled value e while in an `LIO` computation. Expression `label` e_1 e_2 creates a

labeled value, with label e_1, of the result obtained by evaluating the LIO computation e_2. Expression `lFork` e_1 e_2 spawns a thread that computes e_2 and returns a handle with label e_1. Expression `lWait` e inspects the value returned by the spawned computation whose result is accessed by the handle e. Creating, reading, and writing labeled MVars are respectively captured by expressions `newLMVar`, `takeLMVar`, and `putLMVar`.

We consider standard types for Booleans (`Bool`), unit (`()`), and function ($\tau \to \tau$) values. Type ℓ describes security labels. Type `Result` ℓ τ denotes handles used to access labeled results produced by spawned computations, where the results are of type τ and labeled with labels of type ℓ. Type `LMVar` ℓ τ describes labeled `MVars`, with labels of type ℓ and storing values of type τ. Type `LIO` ℓ τ represents monadic LIO computations, with a result type τ and the security labels of type ℓ.

As in [40], we consider that each thread has a thread-local runtime environment σ, which contains the current label σ.`lbl` and the current clearance σ.`clr`. The global environment Σ, common to every thread, holds the global memory store ϕ, which is a mapping from `LMVar` names to `Lb` nodes.

The relation $\langle \Sigma, \langle \sigma, e \rangle \rangle \xrightarrow{\gamma} \langle \Sigma', \langle \sigma', e' \rangle \rangle$ represents a single execution step from thread e, under the run-time environments Σ and σ, to thread e' and run-time environments Σ' and σ'. (This relation does not account for the effects of the cache.) We say that e reduces to e' in one step. Symbol γ ranges over the *internal* events triggered by threads. We utilize internal events to communicate between the threads and the scheduler, e.g., when spawning new threads.

We show the most relevant rules for $\xrightarrow{\gamma}$ in Figure 8. Rule (LAB) generates a labeled value if and only if the label is between the current label and clearance of the LIO computation. Rule (UNLAB) requires that, when the content of a labeled value is "retrieved" and used in a LIO computation, the current label is raised ($\sigma' = \sigma[\texttt{lbl} \mapsto l']$, where $l' = \sigma.\texttt{lbl} \sqcup l$), thus capturing the fact that the remaining computation might depend on e. Rule (LFORK) allows for the creation of a thread and generates the internal event `fork`(e'), where e' is the computation to spawn. The rule allocates a new `LMVar` in order to store the result produced by the spawned thread ($e \gg\!= \lambda x.\texttt{putLMVar}\ m\ x$). Using that `LMVar`, the rule provides a handle to access to the thread's result (`return` (`R` m)). Rule (LWAIT) simply uses the `LMVar` for the handle. Rule (NLMVAR) describes the creation of a new `LMVar` with a label bounded by the current label and clearance ($\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr}$). Rule (TLMVAR) raises the current label ($\sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l]$) when emptying ($\Sigma.\phi[m \mapsto \texttt{Lb}\ l\ \boxdot]$) its content ($\Sigma.\phi(m) = \texttt{Lb}\ l\ e$). Similarly, considering the security level l of a `LMVar`, rule (PLMVAR) raises the current label ($\sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l]$) when filling ($\Sigma.\phi[m \mapsto \texttt{Lb}\ l\ e]$) its content ($\Sigma.\phi(m) = \texttt{Lb}\ l\ \boxdot$). Note that both `takeLMVar` and `putLMVar` observe if the `LMVar` is empty in order to proceed

$$\text{(LAB)}\quad \frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr}}{\langle \Sigma, \langle \sigma, E[\texttt{label}\ l\ e]\rangle\rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{return}\ (\texttt{Lb}\ l\ e)]\rangle\rangle}$$

$$\text{(UNLAB)}\quad \frac{l' = \sigma.\texttt{lbl} \sqcup l \qquad l' \sqsubseteq \sigma.\texttt{clr} \qquad \sigma' = \sigma[\texttt{lbl} \mapsto l']}{\langle \Sigma, \langle \sigma, E[\texttt{unlabel}\ (\texttt{Lb}\ l\ e)]\rangle\rangle \longrightarrow \langle \Sigma, \langle \sigma', E[\texttt{return}\ e]\rangle\rangle}$$

$$\text{(LFORK)}\quad \frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ \boxdot]] \qquad e' = e \gg\!= \lambda x.\texttt{putLMVar}\ m\ x \qquad m \text{ fresh}}{\langle \Sigma, \langle \sigma, E[\texttt{lFork}\ l\ e]\rangle\rangle \xrightarrow{\texttt{fork}(e')} \langle \Sigma', \langle \sigma, E[\texttt{return}\ (\texttt{R}\ m)]\rangle\rangle}$$

$$\text{(LWAIT)}\quad \langle \Sigma, \langle \sigma, E[\texttt{lWait}\ (\texttt{R}\ m)]\rangle\rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{takeLMVar}\ m]\rangle\rangle$$

$$\text{(NLMVAR)}\quad \frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ e]] \qquad m \textit{ fresh}}{\langle \Sigma, \langle \sigma, E[\texttt{newLMVar}\ l\ e]\rangle\rangle \longrightarrow \langle \Sigma', \langle \sigma, E[\texttt{return}\ m]\rangle\rangle}$$

$$\text{(TLMVAR)}\quad \frac{\Sigma.\phi(m) = \texttt{Lb}\ l\ e \qquad e \neq \boxdot \qquad \sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l] \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ \boxdot]]}{\langle \Sigma, \langle \sigma, E[\texttt{takeLMVar}\ m]\rangle\rangle \longrightarrow \langle \Sigma', \langle \sigma', E[\texttt{return}\ e]\rangle\rangle}$$

$$\text{(PLMVAR)}\quad \frac{\Sigma.\phi(m) = \texttt{Lb}\ l\ \boxdot \qquad \sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l] \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ e]]}{\langle \Sigma, \langle \sigma, E[\texttt{putLMVar}\ m\ e]\rangle\rangle \longrightarrow \langle \Sigma', \langle \sigma', E[\texttt{return}\ ()]\rangle\rangle}$$

Fig. 8. Semantics for expressions.

to modify its content. Precisely, `takeLMVar` and `putLMVar` perform a read and a write of the mutable location. Operations on `LMVar` are *bi-directional* and consequently the rules (TLMVAR), and (PLMVAR) require not only that the label of the mentioned `LMVar` be between the current label and current clearance of the thread ($\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr}$), but that the current label be raised appropriately.

1.1 Cache-aware semantics using instruction-based scheduling

Figure 9 presents cache-aware reduction rules for concurrent execution using instruction-based scheduling. The configurations for this relation are very similar to the ones for time-based scheduling in Figure 6 except

that we use an instruction budget b rather than a time quantum q. We write b_i for the initial budget for threads.

The main difference between these semantics and the time-based ones is the cache-aware transition rule (STEP-CA). In this rule, the number of cycles k that the current instruction takes is ignored by the scheduler, counting as one instruction regardless of the time its execution took.

$$\text{(STEP-CA)}\quad \frac{\langle \Sigma, \langle \sigma, e\rangle\rangle_\zeta \longrightarrow_k \langle \Sigma', \langle \sigma', e'\rangle\rangle_{\zeta'} \qquad q > 0}{\langle \Sigma, \zeta, b, \langle \sigma, e\rangle \lhd t_s\rangle \hookrightarrow \langle \Sigma', \zeta', b+1, \langle \sigma', e'\rangle \lhd t_s\rangle}$$

$$\text{(PREEMPT-CA)}\quad \frac{q \le 0}{\langle \Sigma, \zeta, b, t \lhd t_s\rangle \hookrightarrow \langle \Sigma', \zeta, b_i, t_s \rhd t\rangle} \qquad \text{(NO-STEP-CA)}\quad \frac{\langle \Sigma, t\rangle_\zeta \not\longrightarrow \qquad t = \langle \sigma, e\rangle \qquad e \neq v}{\langle \Sigma, \zeta, b, t \lhd t_s\rangle \hookrightarrow \langle \Sigma, \zeta, b_i, t_s \rhd t\rangle}$$

$$\text{(FORK-CA)}\quad \frac{\langle \Sigma, t\rangle_\zeta \xrightarrow{\texttt{fork}(e)}_k \langle \Sigma', \langle \sigma, e'\rangle\rangle_{\zeta'} \qquad t_{\text{new}} = \langle \sigma, e\rangle \qquad q > 0}{\langle \Sigma, \zeta, b, t \lhd t_s\rangle \hookrightarrow \langle \Sigma', \zeta', b+1, \langle \sigma, e'\rangle \lhd t_s \rhd t_{\text{new}}\rangle}$$

$$\text{(EXIT-CA)}\quad \frac{\langle \Sigma, t\rangle_\zeta \longrightarrow_k \langle \Sigma', \langle \sigma, v\rangle\rangle_{\zeta'} \qquad b > 0}{\langle \Sigma, \zeta, b, t \lhd t_s\rangle \hookrightarrow \langle \Sigma', \zeta', b_i, t_s\rangle}$$

Fig. 9. Semantics for threadpools under round-robin instruction-based scheduling

1.2 Security guarantees

In this section, we show that LIO computations satisfy termination-sensitive non-interference. As in [23, 34, 39], we prove this property by using the *term erasure* technique. The erasure function ε_L rewrites data at security levels that the attacker cannot observe into the syntax node $\bullet$.

The function ε_L is defined in such a way that $\varepsilon_L(e)$ contains no information above level L, i.e., the function ε_L replaces all the information more sensitive than L in e with a hole ($\bullet$). In most of the cases, the erasure function is simply applied homomorphically (e.g., $\varepsilon_L(e_1\ e_2) = \varepsilon_L(e_1)\ \varepsilon_L(e_2)$). For `run` expressions, the erasure function is mapped into all threads; all threads with a current label above L are removed from the pool (filter $(\lambda\langle\sigma, e\rangle.e \not\equiv \bullet)$ (map $\varepsilon_L\ t_s$), where $\equiv$ denotes syntactic equivalence). The computation performed in a certain sequential configuration

is erased if the current label is above L. For runtime environments and stores, we map the erasure function into their components. Similarly, a labeled value is erased if the label assigned to it is above L.

Following the definition of the erasure function, we introduce a new evaluation relation $\longrightarrow_L$ as follows:

$$\frac{\langle \Sigma, \langle \sigma, t\rangle\rangle \longrightarrow \langle \Sigma', \langle \sigma', t\rangle\rangle}{\langle \Sigma, \langle \sigma, t\rangle\rangle \longrightarrow_L \varepsilon_L(\langle \Sigma', \langle \sigma', t'\rangle\rangle)}$$

The relation $\longrightarrow_L$ guarantees that confidential data, i.e., data not below level L, is erased as soon as it is created. We write $\longrightarrow_L^*$ for the reflexive and transitive closure of $\longrightarrow_L$.

In order to prove non-interference, we will establish a simulation relation between $\longrightarrow^*$ and $\longrightarrow_L^*$ through the erasure function: erasing all secret data and then taking evaluation steps in $\longrightarrow_L$ is equivalent to taking steps in $\longrightarrow$ first, and then erasing all secret values in the resulting configuration. Note that this relation would not hold if information from some level above L was being leaked by the program. In the rest of this section, we only consider well-typed terms to ensure there are no stuck configurations.

We start by showing that the evaluation relation $\longrightarrow_L$ is deterministic

Proposition 1 (Determinacy of $\longrightarrow_L$). *If $\langle \Sigma, t\rangle_\zeta \longrightarrow_L \langle \Sigma', t'\rangle_{\zeta'}$ and $\langle \Sigma, t\rangle_\zeta \longrightarrow_L \langle \Sigma'', t''\rangle_{\zeta''}$, then $\langle \Sigma', t'\rangle_{\zeta'} = \langle \Sigma'', t''\rangle_{\zeta''}$.*

Proof. By induction on expressions and evaluation contexts, showing there is always a unique redex in every step.

The next lemma establishes a simulation between $\hookrightarrow^*$ and $\hookrightarrow_L^*$.

Lemma 1 (Many-step simulation). *If $\langle \Sigma, \zeta, b, t_s\rangle \hookrightarrow^* \langle \Sigma', \zeta', b', t_s'\rangle$, then $\varepsilon_L(\langle \Sigma, \zeta, b, t_s\rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma', \zeta', b', t_s'\rangle)$.*

Proof. In order to prove this result, we rely on properties of the erasure function, such as the fact that it is idempotent and homomorphic to the application of evaluation contexts and substitution. We show that the result holds by case analysis on the rule used to derive $\langle \Sigma, t_s\rangle \hookrightarrow^* \langle \Sigma', t_s'\rangle$, and considering different cases for threads whose current label is below (or not) level L.

The L-equivalence relation $\approx_L$ is an equivalence relation between configurations (and their parts), defined as the equivalence kernel of the erasure function ε_L: $\langle \Sigma, \zeta, b, t_s\rangle \approx_L \langle \Sigma', \zeta', b', r_s\rangle$ iff $\varepsilon_L(\langle \Sigma, \zeta, b, t_s\rangle) = \varepsilon_L(\langle \Sigma', \zeta', b', r_s\rangle)$. If two configurations are L-equivalent, they agree on all data below or at level L, i.e., they cannot be distinguished by an attacker at level L. Note that two queues are L-equivalent iff the threads

with current label no higher than L are pairwise L-equivalent in the order that they appear in the queue.

The next theorem shows the non-interference property. It essentially states that if we take two executions of a program with two L-equivalent inputs, then for every intermediate step of the computation of the first run, there is a corresponding step in the computation of the second run which results in an L-equivalent configuration.

Theorem 2 (Termination-sensitive non-interference). *Given a computation e (with no* `Lb`*,* $()^{\mathtt{LIO}}$*,* $\boxdot$*,* `R`*, and* $\bullet$*) where* $\Gamma \vdash e : \mathtt{Labeled}\ \ell\ \tau \rightarrow \mathtt{LIO}\ \ell\ (\mathtt{Labeled}\ \ell\ \tau')$*, an attacker at level L, an initial securiy context σ, runtime environments Σ_1 and Σ_2 where $\Sigma_1.\phi = \Sigma_2.\phi = \varnothing$, and initial cache states ζ_1 and ζ_2, then*

$$\begin{aligned} &\forall e_1 e_2.(\Gamma \vdash e_i : \mathtt{Labeled}\ \ell\ \tau)_{i=1,2} \wedge\ e_1 \approx_L e_2 \\ &\qquad \wedge \langle \Sigma_1, \zeta_1, b_i, \langle \sigma, e\ e_1 \rangle \rangle \hookrightarrow^* \langle \Sigma_1', \zeta_1', b_1', t_s^1 \rangle \\ \Rightarrow\ &\exists \Sigma_2' \zeta_2' b_2' t_s^2 . \langle \Sigma_2, \zeta_2, b_i, \langle \sigma, e\ e_2 \rangle \rangle \hookrightarrow^* \langle \Sigma_2', \zeta_2', b_2', t_s^2 \rangle \\ &\qquad \wedge \langle \Sigma_1', \zeta_1', b_1', t_s^1 \rangle \approx_L \langle \Sigma_2', \zeta_2', b_2', t_s^2 \rangle \end{aligned}$$

Proof. Take $\langle \Sigma_1, \zeta_1, b_i, \langle \sigma, e\ e_1 \rangle \rangle \hookrightarrow^* \langle \Sigma_1', \zeta_1', b_1', t_s^1 \rangle$ and apply Lemma 1 to get $\varepsilon_L(\langle \Sigma_1, \zeta_1, b_i, \langle \sigma, e\ e_1 \rangle \rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma_1', \zeta_1', b_1', t_s^1 \rangle)$. We know this reduction only includes public ($\sqsubseteq L$) steps, so the number of steps is lower than or equal to the number of steps in the first reduction.

We can always find a reduction starting from $\varepsilon_L(\langle \Sigma_2, \zeta_2, b_i, \langle \sigma, e\ e_2 \rangle \rangle)$ with the same number of steps as $\varepsilon_L(\langle \Sigma_1, \zeta_1, b_i, \langle \sigma, e\ e_1 \rangle \rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma_1', \zeta_1', b_1', t_s^1 \rangle)$, so by the Determinacy Lemma we have $\varepsilon_L(\langle \Sigma_2, \zeta_2, b_i, \langle \sigma, e\ e_2 \rangle \rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma_2', \zeta_2', b_2', t_s^2 \rangle)$. By Lemma 1 again, we get $\langle \Sigma_2, \zeta_2, b_i, \langle \sigma, e\ e_2 \rangle \rangle \hookrightarrow^* \langle \Sigma_2', \zeta_2', b_2', t_s^2 \rangle$ and therefore $\langle \Sigma_1', \zeta_1', b_1', t_s^1 \rangle \approx_L \langle \Sigma_2', \zeta_2', b_2', t_s^2 \rangle$.

CHAPTER

FOUR

A LIBRARY FOR REMOVING CACHE-BASED ATTACKS IN CONCURRENT INFORMATION FLOW SYSTEMS

Pablo Buiras, Amit Levy, Deian Stefan
Alejandro Russo, David Mazières

Abstract. Information-flow control (IFC) is a security mechanism conceived to allow untrusted code to manipulate sensitive data without compromising confidentiality. Unfortunately, untrusted code might exploit some covert channels in order to reveal information. In this paper, we focus on the LIO concurrent IFC system. By leveraging the effects of hardware caches (e.g., the CPU cache), LIO is susceptible to attacks that leak information through the *internal timing covert channel*. We present a *resumption*-based approach to address such attacks. Resumptions provide fine-grained control over the interleaving of thread computations at the library level. Specifically, we remove cache-based attacks by enforcing that every thread yield after executing an "instruction," i.e., atomic action. Importantly, our library allows for porting the full LIO library—our resumption approach handles local state and exceptions, both features present in LIO. To amend for performance degradations due to the library-level thread scheduling, we provide two novel primitives. First, we supply a primitive for securely executing pure code in parallel. Second, we provide developers a primitive for controlling the granularity of "instructions"; this allows developers to adjust the frequency of context switching to suit application demands.

1 Introduction

Popular website platforms, such as Facebook, run third-party applications (apps) to enhance the user experience. Unfortunately, in most of today's platforms, once an app is installed it is usually granted full or partial access to the user's sensitive data—the users have no guarantees that their data is not arbitrarily ex-filtrated once apps are granted access to it [18]. As demonstrated by Hails [9], information-flow control (IFC) addresses many of these limitations by restricting how sensitive data is disseminated. While promising, IFC systems are not impervious to attacks; the presence of *covert channels* allows attackers to leak sensitive information.

Covert channels are mediums not intended for communication, which nevertheless can be used to carry and, thus, reveal information [19]. In this work, we focus on the *internal timing covert channel* [33]. This channel emanates from the mere presence of concurrency and shared resources. A system is said to have an internal timing covert channel when an attacker, as to reveal sensitive data, can alter *the order of public events* by affecting the timing behavior of threads. To avoid such attacks, several authors propose decoupling computations manipulating sensitive data from those writing into public resources (e.g., [4, 5, 27, 30, 35]).

Decoupling computations by security levels only works when all shared resources are modeled. Similar to most IFC systems, the concurrent IFC system LIO [35] only models shared resources at the programming language level and does not explicitly consider the effects of hardware. As shown in [37], LIO threads can exploit the underlying CPU cache to leak information through the internal timing covert channel.

We propose using *resumptions* to model interleaved computations. (We refer the interested reader to [10] for an excellent survey of resumptions.) A resumption is either a (computed) value or an atomic action which, when executed, returns a new resumption. By expressing thread computations as a series of resumptions, we can leverage resumptions for controlling concurrency. Specifically, we can interleave atomic actions, or "instructions," from different threads, effectively forcing each thread to yield at deterministic points. This ensures that scheduling is not influenced by underlying caches and thus cannot be used to leak secret data. We address the attacks on the recent version of LIO [35] by implementing a Haskell library which ports the LIO API to use resumptions. Since LIO threads possess local state and handle exceptions, we extend resumptions to account for these features.

In principle, it is possible to force deterministic interleaving by means other than resumptions; in [37] we show an instruction-based scheduler that achieves this goal. However, Haskell's monad abstraction allows us to to easily model resumptions as a library. This has two consequences. First, and different from [37], it allows us to deploy a version of LIO

that does not rely on changes to the Haskell compiler. Importantly, LIO's concurrency primitives can be modularly redefined, with little effort, to operate on resumptions. Second, by effectively implementing "instruction based-scheduling" at the level of library primitives, we can address cache attacks not covered by the approach described in [37] (see Section 5).

In practice, a library-level interleaved model of computations imposes performance penalties. With this in mind, we provide primitives that allow developers to execute code in parallel, and means for securely controlling the granularity of atomic actions (which directly affects performance).

Although our approach addresses internal timing attacks in the presence of shared hardware, the library suffers from leaks that exploit the termination channel, i.e., programs can leak information by not terminating. However, this channel can only be exploited by brute-force attacks that leak data external to the program—an attacker cannot leak data within the program, as can be done with the internal timing covert channel.

2 Cache Attacks on Concurrent IFC Systems

Figure 1 shows an attack that leverages the timing effects of the underlying cache in order to leak information through the internal timing covert channel. In isolation, all three threads are secure. However, when executed concurrently, threads B and C race to write to a public, shared variable `l`. Importantly, the race outcome depends on the state of the secret variable `h`, by changing the contents of underlying CPU cache according to its value (e.g., by creating and traversing a large array as to fill the cache with new data).

The attack proceeds as follows. First, thread A fills the cache with the contents of a public array `lowArray`. Then, depending on the secret variable `h`, it evicts data from the cache (by filling it with arbitrary data)

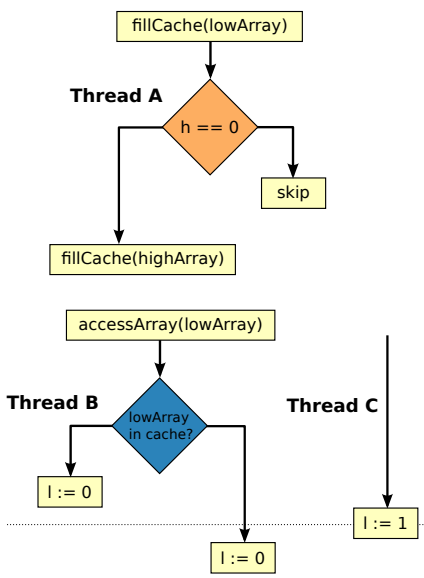

Fig. 1. Cache attack

or leaves it intact. Concurrently,
public threads B and C delay execution long enough for A to finish. Subsequently, thread B accesses elements of the public array `lowArray`, and writes 0 to public variable `l`; if the array has been evicted from the cache (`h==0`), the amount of time it takes to perform the read, and thus the write to `l`, will be much longer than if the array is still in the cache. Hence, to leak the value of `h`, thread C simply needs to delay writing 1 to `l` long enough so that it is above the case where the cache is full (with the public array), but shorter than it take to refill the cache with the (public) array. Observing the contents of `l`, the attacker directly learns the value of `h`.

This simple attack has previously been demonstrated in [37], where confidential data from the GitStar system [9], build atop LIO, was leaked. Such attacks are not limited to LIO or IFC systems; cache-based attacks against many system, including cryptographic primitives (e.g., RSA and AES), are well known [1, 23, 26, 40].

The next section details the use of resumptions in modeling concurrency at the programming language level by defining atomic steps, which are used as the thread scheduling quantum unit. By scheduling threads according to the number of executed atoms, the attack in Figure 1 is eliminated. As in [37], this is the case because an atomic step runs till completion, regardless of the state of the cache. Hence, the timing behavior of thread B, which was previously leaked to thread C by the time of preemption, is no longer disclosed. Specifically, the scheduling of thread C's `l:=1` does not depend on the time it takes thread B to read the public array from the cache; rather it depends on the atomic actions, which do not depend on the cache state. In addition, our use of resumptions also eliminates attacks that exploit other timing perturbations produced by the underlying hardware, e.g., TLB misses, CPU bus contention, etc.

3 Modeling Concurrency with Resumptions

In pure functional languages, computations with side-effects are encoded as values of abstract data types called *monads* [22]. We use the type $m\ a$ to denote computations that produce results of type a and may perform side-effects in monad m. Different side-effects are often handled by different monads. In Haskell, there are monads for performing inputs and outputs (monad IO), handling errors (monad $Error$), etc. The IFC system LIO simply exposes a monad, LIO, in which security checks are performed before any IO side-effecting action.

Resumptions are a simple approach to modeling interleaved computations of concurrent programs. A resumption, which has the form $res ::= x \mid \alpha \triangleright res$, is either a computed value x or an atomic action α fol-

```
data Thread m a where
  Done :: a → Thread m a
  Atom :: m (Thread m a)
       → Thread m a
  Fork :: Thread m ()
       → Thread m a
       → Thread m a
```

Fig. 2. Threads as Resumptions

```
sch :: [Thread m ()] → m ()
sch []                  = return ()
sch ((Done _) : thrds)  = sch thrds
sch ((Atom m) : thrds) =
  do res ← m; sch (thrds ++ [res])
sch ((Fork res res') : thrds) =
  sch ((res : thrds) ++ [res'])
```

Fig. 3. Simple round-robin scheduler

lowed by a new resumption res. Using this notion, we can break down a program that is composed of a series of instructions into a program that executes an atomic action and yields control to a scheduler by giving it its subsequent resumption. For example, program $P := i_1; i_2; i_3$, which performs three side-effecting instructions in sequence, can be written as $res_P := i_1; i_2 \triangleright i_3 \triangleright ()$, where $()$ is a value of a type with just one element, known as *unit*. Here, an atomic action α is any sequence of instructions. When executing res_P, instructions i_1 and i_2 execute atomically, after which it yields control back to the scheduler by supplying it the resumption $res'_P := i_3 \triangleright ()$. At this point, the scheduler may schedule atomic actions from other threads or execute res'_P to resume the execution of P. Suppose program $Q := j_1; j_2$, rewritten as $j_1 \triangleright j_2 \triangleright ()$, runs concurrently with P. Our concurrent execution of P and Q can be modeled with resumptions, under a round-robin scheduler, by writing it as $P||Q := i_1; i_2 \triangleright j_1 \triangleright i_3 \triangleright j_2 \triangleright () \triangleright ()$. In other words, resumptions allow us to implement a scheduler that executes $i_1; i_2$, postponing the execution of i_3, and executing atomic actions from Q in the interim.

Implementing threads as resumptions As previously done in [10, 11], Fig. 2 defines threads as resumptions at the programming language level. The thread type $(Thread\ m\ a)$ is parametric in the resumption computation value type (a) and the monad in which atomic actions execute (m)[1]. (Symbol :: introduces type declarations and $\rightarrow$ denotes function types.) The definition has several value constructors for a thread. Constructor $Done$ captures computed values; a value $Done\ a$ represents the computed value a. Constructor $Atom$ captures a resumption of the form $\alpha \triangleright res$. Specifically, $Atom$ takes a monadic action of type $m\ (Thread\ m\ a)$, which denotes an atomic computation in monad m that returns a new resumption as a result. In other words, $Atom$ captures both the atomic action that is being executed (α) and the subsequent resumption (res). Finally, constructor $Fork$ captures the action of spawning new threads; value $Fork\ res\ res'$ encodes a computation wherein a new thread runs

[1] In our implementation, atomic actions α (as referred as in $\alpha \triangleright res$) are actions described by the monad m.

resumption res and the original thread continues as res'.[2] As in the standard Haskell libraries, we assume that a fork does not return the new thread's final value and thus the type of the new thread/resumption is simply $Thread\ m\ ()$.

Programming with resumptions Users do not build programs based on resumptions by directly using the constructors of $Thread\ m\ a$. Instead, they use the interface provided by Haskell monads:
$return :: a \rightarrow Thread\ m\ a$ and $(\gg\!=) :: Thread\ m\ a \rightarrow (a \rightarrow Thread\ m\ b) \rightarrow Thread\ m\ b$. The expression $return\ a$ creates a resumption which consists of the computed value a, i.e., it corresponds to $Done\ a$. The operator $(\gg\!=)$, called *bind*, is used to sequence atomic computations. Specifically, the expression $res \gg\!= f$ returns a resumption that consists of the execution of the atomic actions in res followed by the atomic actions obtained from applying f to the result produced by res. We sometimes use Haskell's **do**-notation to write such monadic computations. For example, the expression $res \gg\!= (\lambda a \rightarrow return\ (a+1))$, i.e., actions described by the resumption res followed by $return\ (a+1)$ where a is the result produced by res, is written as **do** $a \leftarrow res; return\ (a+1)$.

Scheduling computations We use round-robin to schedule atomic actions of different threads. Fig. 3 shows our scheduler implemented as a function from a list of threads into an interleaved computation in the monad m. The scheduler behaves as follows. If there is an empty list of resumptions, the scheduler, and thus the program, terminates. If the resumption at the head of the list is a computed value ($Done\ \_$), the scheduler removes it and continues scheduling the remaining threads (**sch** $thrds$). (Recall that we are primarily concerned with the side-effects produced by threads and not about their final values.) When the head of the list is an atomic step ($Atom\ m$), **sch** runs it ($res \leftarrow m$), takes the resulting resumption (res), and appends it to the end of the thread list (**sch** $(thrds +\!\!+ [res])$). Finally, when a thread is forked, i.e., the head of the list is a $Fork\ res\ res'$, the spawned resumption is placed at the front of the list ($res : thrds$). Observe that in both of the latter cases the scheduler is invoked recursively—hence we keep evaluating the program until there are no more threads to schedule. We note that although we choose a particular, simple scheduling approach, our results naturally extend for a wide class of deterministic schedulers [28, 38].

4 Extending Resumptions with State and Exceptions

LIO provides general programming language abstrations (e.g., state and exceptions), which our library must preserve to retain expressiveness. To

[2] Spawning threads could also be represented by a equivalent constructor $Fork' :: Thread\ m\ () \rightarrow Thread\ m\ a$, we choose $Fork$ for pedagogical reasons.

this end, we extend the notion of resumptions and modify the scheduler to handle thread local state and exceptions.

Thread local state As described in [34], the LIO monad keeps track of a *current label*, L_{cur}. This label is an upper bound on the labels of all data in lexical scope. When a computation C, with current label L_C, observes an object labeled L_O, C's label is raised to the least upper bound or *join* of the two labels, written $L_C \sqcup L_O$. Importantly, the current label governs where the current computation can write, what labels may be used when creating new channels or threads, etc. For example, after reading an object O, the computation should not be able to write to a channel K if L_O is more confidential than L_K—this would potentially leak sensitive information (about O) into a less sensitive channel. We write $L_C \sqsubseteq L_K$ when L_K at least as confidential as L_C and information is allowed to flow from the computation to the channel.

$$
\begin{aligned}
&\mathbf{sch}\ ((Atom\ m) : thrds) = \\
&\quad \mathbf{do}\ res \leftarrow m \\
&\qquad\ st\ \ \leftarrow get \\
&\qquad \mathbf{sch}\ (thrds +\!\!+ [put\ st \succ res]) \\
&\mathbf{sch}\ ((Fork\ res\ res') : thrds) = \\
&\quad \mathbf{do}\ st \leftarrow get \\
&\qquad \mathbf{sch}\ ((res : thrds) +\!\!+ [put\ st \succ res'])
\end{aligned}
$$

Fig. 4. Context-switch of local state

Using our resumption definition of Section 3, we can model concurrent LIO programs as values of type $Thread\ LIO$. Unfortunately, such programs are overly restrictive—since LIO threads would be sharing a single current label—and do not allow for the implementation of many important applications. Instead, and as done in the concurrent version of LIO [35], we track the state of each thread, independently, by modifying resumptions, and the scheduler, with the ability to context-switch threads with state.

Figure 4 shows these changes to **sch**. The context-switching mechanism relies on the fact that monad m is a state monad, i.e., provides operations to retrieve (get) and set (put) its state. LIO is a state monad,[3] where the state contains (among other things) L_{cur}. Operation $(\succ) :: m\ b \to Thread\ m\ a \to Thread\ m\ a$ modifies a resumption in such a way that its first atomic step ($Atom$) is extended with $m\ b$ as the first action. Here, $Atom$ consists of executing the atomic step ($res \leftarrow m$), taking a snapshot of the state ($st \leftarrow get$), and restoring it when executing the thread again ($put\ st \succ res$). Similarly, the case for $Fork$ saves the state before creating the child thread and restores it when the parent thread executes again ($put\ st \succ res'$).

[3] For simplicity of exposition, we use get and set. However, LIO only provides such functions to trusted code. In fact, the monad LIO is not an instance of $MonadState$ since this would allow untrusted code to arbitrarily modify the current label—a clear security violation.

Exception handling As described in [36], LIO provides a secure way to throw and catch exceptions—a feature crucial to many real-world applications. Unfortunately, simply using LIO's *throw* and *catch* as atomic actions, as in the case of local state, results in non-standard behavior. In particular, in the interleaved computation produced by **sch**, an atomic action from a thread may throw an exception that would propagate outside the thread group and crash the program. Since we do not consider leaks due to termination, this does not impact security; however, it would have non-standard and restricted semantics. Hence, we first extend our scheduler to introduce a top-level *catch* for every spawned thread.

Besides such an extension, our approach still remains quite limiting. Specifically, LIO's *catch* is defined at the level of the monad *LIO*, i.e., it can only be used inside atomic steps. Therefore, catch-blocks are prevented from being extended beyond atomic actions. To address this limitation, we lift exception handling to work at the level of resumptions.

To this end, we consider a monad m that handles exceptions, i.e., a monad for which $throw :: e \to m\ a$ and $catch :: m\ a \to (e \to m\ a) \to m\ a$, where e is a type denoting exceptions, are accordingly defined. Function *throw* throws the exception supplied as an argument. Function *catch* runs the action supplied as the first argument ($m\ a$), and if an exception is thrown, then executes the handler ($e \to m\ a$) with the value of the exception passed as an argument. If no exceptions are raised, the result of the computation (of type a) is simply returned.

```
throw e = Atom (LIO.throw e)
catch (Done a) _ = Done a
catch (Atom a) handler =
  Atom (LIO.catch
    (do res ← a
        return (catch res handler))
    (λe → return (handler e)))
catch (Fork res res') handler =
  Fork res (catch res' handler)
```

Fig. 5. Exception handling for resumptions

Figure 5 shows the definition of exception handling for resumptions. Since *LIO* defines *throw* and *catch* [36], we qualify these underlying functions with *LIO* to distinguish them from our resumption-level *throw* and *catch*. When throwing an exception, the resumption simply executes an atomic step that throws the exception in *LIO* ($LIO.throw\ e$).

The definitions of *catch* for *Done* and *Fork* are self explanatory. The most interesting case for *catch* is when the resumption is an *Atom*. Here, *catch* applies $LIO.catch$ step by step to each atomic action in the sequence; this is necessary because exceptions can only be caught in the *LIO* monad. As shown in Fig. 5, if no exception is thrown, we simply return the resumption produced by m. Conversely, if an exception is

raised, $LIO.catch$ will trigger the exception handler which will return a resumption by applying the top-level $handler$ to the exception e. To clarify, consider catching an exception in the resumption $\alpha_1 \triangleright \alpha_2 \triangleright x$. Here, $catch$ executes α_1 as the first atomic step, and if no exception is raised, it executes α_2 as the next atomic step; on the other hand, if an exception is raised, the resumption $\alpha_2 \triangleright x$ is discarded and $catch$, instead, executes the resumption produced when applying the exception handler to the exception.

5 Performance Tuning

Unsurprisingly, interleaving computations at the library-level introduces performance degradation. To alleviate this, we provide primitives that allow developers to control the granularity of atomic steps—fine-grained atoms allow for more flexible programs, but also lead to more context switches and thus performance degradation (as we spend more time context switching). Additionally, we provide a primitive for the parallel execution of pure code. We describe these features—which do not affect our security guarantees—below.

Granularity of atomic steps To decrease the frequency of context switches, programmers can treat a complex set of atoms (which are composed using monadic bind) as a single atom using $singleAtom :: Thread\ m\ a \rightarrow Thread\ m\ a$. This function takes a resumption and "compresses" all its atomic steps into one. Although $singleAtom$ may seem unsafe, e.g., because we do not restrict threads from adjust the granularity of atomic steps according to secrets, in Section 6 we show that this is not the case—it is the atomic execution of atoms, regardless of their granularity, that ensures security.

Parallelism As in [37], we cannot run one scheduler **sch** per core to gain performance through parallelism. Threads running in parallel can still race to public resources, and thus vulnerable to internal timing attacks (that may, for example, rely on the L3 CPU cache). In principle, it is possible to securely parallelize arbitrary side-effecting computations if races (or their outcomes) to shared public resource are eliminated. Similar to *observational low-determinism* [41], our library could allow parallel computations to compute on disjoint portions of the memory. However, whenever side-effecting computations follow parallel code, we would need to impose synchronization barriers to enforce that all side-effects are performed in a pre-determined order. It is precisely this order, and LIO's safe side-effecting primitives for shared-resources, that hides the outcome of any potential dangerous parallel race. In this paper, we focus on executing pure code in parallel; we leave side-effecting code to future work.

Pure computations, by definition, cannot introduce races to shared resources since they do not produce side effects.[4] To consider such computations, we simply extend the definition of $Thread$ with a new constructor: $Parallel :: pure\ b \rightarrow (b \rightarrow Thread\ m\ a) \rightarrow Thread\ m\ a$. Here, $pure$ is a monad that characterizes pure expressions, providing the primitive $runPure :: pure\ b \rightarrow b$ to obtain the value denoted by the code given as argument. The monad $pure$ could be instantiated to Par, a monad that parallelizes pure computations in Haskell [21], with $runPure$ set to $runPar$. In a resumption, $Parallel\ p\ f$ specifies that p is to be executed in a separate Haskell thread—potentially running on a different core than the interleaved computation. Once p produces a value x, f is applied to x to produce the next resumption to execute.

$$
\begin{array}{l}
\textbf{sch}\ (Parallel\ p\ f : thrds) = \\
\quad \textbf{do}\ res \leftarrow sync\ (\lambda v \rightarrow putMVar\ v\ (runPure\ p)) \\
\qquad\qquad\qquad\quad (\lambda v \rightarrow takeMVar\ v) \\
\qquad\qquad\qquad\quad f \\
\qquad \textbf{sch}\ (thrds +\!\!+ [res])
\end{array}
$$

Fig. 6. Scheduler for parallel computations

Figure 6 defines **sch** for pure computations, where interaction between resumptions and Haskell-threads gets regulated. The scheduler relies on well-established synchronization primitives called MVars [13]. A value of type $MVar$ is a mutable location that is either empty or contains a value. Function $putMVar$ fills the MVar with a value if it is empty and blocks otherwise. Dually, $takeMVar$ empties an MVar if it is full and returns the value; otherwise it blocks. Our scheduler implementation **sch** simply takes the resumption produced by the $sync$ function and schedules it at the end of the thread pool. Function $sync$, internally creates a fresh MVar v and spawns a new Haskell-thread to execute $putMVar\ v\ (runPure\ p)$. This action will store the result of the parallel computation in the provided MVar. Subsequently, $sync$ returns the resumption res, whose first atomic action is to read the parallel computation's result from the MVar ($takeMVar\ v$). At the time of reading, if a value is not yet ready, the atomic action will block the whole interleaved computation. However, once a value x is produced (in the separate thread), f is applied to it and the execution proceeds with the produced resumption ($f\ x$).

[4] In the case of Haskell, lazy evaluation may pose a challenge since whether or not a thunk has been evaluated is indeed an effect on a cache [24]. Though our resumption-based approach handles this for the single-core case, handling this in general is part of our ongoing work.

(DONE)

$$\frac{}{\langle \Sigma, \mathbf{sch}\ (\mathit{Done}\ x : t_s)\rangle \longrightarrow \langle \Sigma, \mathbf{sch}\ t_s\rangle}$$

(ATOM)

$$\frac{\langle \Sigma, m\rangle \longrightarrow^{*} \langle \Sigma', (e)^{\mathtt{LIO}}\rangle}{\langle \Sigma, \mathbf{sch}\ (\mathit{Atom}\ (\mathit{put}\ \Sigma.\mathtt{lbl} \gg m) : t_s)\rangle \longrightarrow \langle \Sigma', \mathbf{sch}\ (t_s +\!\!+ [\mathit{put}\ \Sigma.\mathtt{lbl} \succ e])\rangle}$$

(FORK)

$$\frac{}{\langle \Sigma, \mathbf{sch}\ (\mathit{Fork}\ m_1\ m_2 : t_s)\rangle \longrightarrow \langle \Sigma, \mathbf{sch}\ ((m_1 : t_s) +\!\!+ [\mathit{put}\ \Sigma.\mathtt{lbl} \succ m_2])\rangle}$$

Fig. 7. Semantics for **sch** expressions.

6 Soundness

In this section, we extend the previous formalization of LIO [34] to model the semantics of our concurrency library. We present the syntax extensions that we require to model the behavior of the *Thread* monad:

Expression: $e ::= \ \ldots \mid \mathbf{sch}\, e_s \mid \mathit{Atom}\, e \mid \mathit{Done}\, e \mid \mathit{Fork}\, e\, e \mid \mathit{Parallel}\, e\, e$

where e_s is a list of expressions. For brevity, we omit a full presentation of the syntax and semantics, since we rely on previous results in order to prove the security property of our approach. The interested reader is referred to [6].

Expressions are the usual λ-calculus expressions with special syntax for monadic effects and LIO operations. The syntax node **sch** e_s denotes the scheduler running with the list of threads e_s as its thread pool. The nodes $\mathit{Atom}\ e$, $\mathit{Done}\ e$, $\mathit{Fork}\ e\ e$ and $\mathit{Parallel}\ e\ e$ correspond to the constructors of the *Thread* data type. In what follows, we will use metavariables x, m, p, t, v and f for different kinds of expressions, namely values, monadic computations, pure computations, threads, MVars and functions, respectively.

We consider a global environment Σ which contains the current label of the computation ($\Sigma.\mathtt{lbl}$), and also represents the resources shared among all threads, such as mutable references. We start from the one-step reduction relation[5] $\langle \Sigma, e\rangle \longrightarrow \langle \Sigma', e'\rangle$, which has already been defined for LIO [34]. This relation represents a single evaluation step from e to e', with Σ as the initial environment and Σ' as the final one. Presented as an extension to the $\longrightarrow$ relation, Figure 7 shows the reduction rules for concurrent execution using **sch**. The configurations for this relation are of the form $\langle \Sigma, \mathbf{sch}\ t_s\rangle$, where Σ is a runtime environment and t_s is a list of *Thread* computations. Note that the computation in

[5] As in [35], we consider a version of $\longrightarrow$ which does not include the operation *toLabeled*, since it is susceptible to internal timing attacks.

$$\text{(SEQ)}\quad \frac{\langle \Sigma, e\rangle \longrightarrow \langle \Sigma', e'\rangle \qquad P \Rightarrow P'}{\langle \Sigma, e \parallel P\rangle \hookrightarrow \langle \Sigma', e' \parallel P'\rangle}$$

$$\text{(PURE)}\quad \frac{P \Rightarrow P' \qquad v_s \text{ fresh MVar} \qquad s = \Sigma.\texttt{lbl}}{\begin{array}{c}\langle \Sigma, \mathbf{sch}\ (Parallel\ p\ f : t_s) \parallel P\rangle \hookrightarrow \\ \langle \Sigma, \mathbf{sch}\ (t_s +\!\!+ [Atom\ (takeMVar\ v_s \gg\!= f)]) \parallel P' \parallel (putMVar\ v_s\ (runPure\ p))_s\rangle\end{array}}$$

$$\text{(SYNC)}\quad \frac{P \Rightarrow P'}{\begin{array}{c}\langle \Sigma, \mathbf{sch}\ (Atom\ (takeMVar\ v_s \gg\!= f) : t_s) \parallel (putMVar\ v_s\ x)_s \parallel P\rangle \hookrightarrow \\ \langle \Sigma, \mathbf{sch}\ (f\ x : t_s) \parallel P'\rangle\end{array}}$$

Fig. 8. Semantics for **sch** expressions with parallel processes.

an $Atom$ always begins with either $put\ \Sigma.\texttt{lbl}$ for some label $\Sigma.\texttt{lbl}$, or with $takeMVar\ v$ for some MVar v. Rules (DONE), (ATOM), and (FORK) basically behave like the corresponding equations in the definition of **sch** (see Figures 3 and 4). In rule (ATOM), the syntax node $(e)^{\text{LIO}}$ represents an LIO computation that has produced expression e as its result. Although **sch** applications should expand to their definitions, for brevity we show the unfolding of the resulting expressions into the next recursive call. This unfolding follows from repeated application of basic λ-calculus reductions.

Figure 8 extends relation $\longrightarrow$ into $\hookrightarrow$ to express pure parallel computations. The configurations for this relation are of the form $\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle$, where P is an abstract process representing a pure computation that is performed in parallel. These abstract processes would be reified as native Haskell threads. The operator ($\parallel$), representing parallel process composition, is commutative and associative.

As described in the previous section, when a $Thread$ evaluates a $Parallel$ computation, a new native Haskell thread should be spawned in order to run it. Rule (PURE) captures this intuition. A fresh MVar v_s (where s is the current label) is used for synchronization between the parent and the spawned thread. A process is denoted by $putMVar\ v_s$ followed by a pure expression, and it is also tagged with the security level of the thread that spawned it.

Pure processes are evaluated in parallel with the main threads managed by **sch**. The relation $\Rightarrow$ nondeterministically evaluates one process in a parallel composition and is defined as follows.

$$\frac{runPure\ p \longrightarrow^* x}{(putMVar\ v_s\ (runPure\ p))_s \parallel P \Rightarrow (putMVar\ v_s\ x)_s \parallel P}$$

For simplicity, we consider the full evaluation of one process until it yields a value as just one step, since the computations involved are pure and therefore cannot leak data. Rule (SEQ) in Figure 8 represents steps where no parallel forking or synchronization is performed, so it executes one $\longrightarrow$ step alongside a $\Rightarrow$ step.

Rule (SYNC) models the synchronization barrier technique from Section 5. When an $Atom$ of the form $(takeMVar\ v_s \gg\!= f)$ is evaluated, execution blocks until the pure process with the corresponding MVar v_s completes its computation. After that, the process is removed and the scheduler resumes execution.

Security guarantees We show that programs written using our library satisfy termination-insensitive non-interference, i.e., an attacker at level L cannot distinguish the results of programs that run with indistinguishable inputs . This result has been previously established for the sequential version of LIO [34]. As in [20, 31, 34], we prove this property by using the *term erasure* technique.

In this proof technique, we define function ε_L in such a way that $\varepsilon_L(e)$ contains only information below or equal to level L, i.e., the function ε_L replaces all the information more sensitive than L or incomparable to L in e with a hole ($\bullet$). We adapt the previous definition of ε_L to handle the new constructs in the library. In most of the cases, the erasure function is simply applied homomorphically (e.g., $\varepsilon_L(e_1\ e_2) = \varepsilon_L(e_1)\ \varepsilon_L(e_2)$). For **sch** expressions, the erasure function is mapped into the list; all threads with a current label above L are removed from the pool ($\mathit{filter}\ (\not\equiv \bullet)\ (map\ \varepsilon_L\ t_s)$), where $\equiv$ denotes syntactic equivalence). Analogously, erasure for a parallel composition consists of removing all processes using an MVar tagged with a level not strictly below or equal to L. The computation performed in a certain $Atom$ is erased if the label is not strictly below or equal than L. This is given by

$$\varepsilon_L(Atom\ (put\ s \gg m)) = \begin{cases} \bullet & , \ s \not\sqsubseteq L \\ put\ s \gg \varepsilon_L\ (m) & , \ \text{otherwise} \end{cases}$$

A similar rule exists for expressions of the form $Atom\ (takeMVar\ v_s \gg\!= f)$. Note that this relies on the fact that an atom must be of the form $Atom\ (put\ s \gg m)$ or $Atom\ (takeMVar\ v_s \gg\!= f)$ by construction. For expressions of the form $Parallel\ p\ f$, erasure behaves homomorphically, i.e. $\varepsilon_L(Parallel\ p\ f) =$
$Parallel\ \varepsilon_L(p)\ (\varepsilon_L \circ f)$.

Following the definition of the erasure function, we introduce the evaluation relation $\hookrightarrow_L$ as follows: $\langle \Sigma, t \parallel P\rangle \hookrightarrow_L \varepsilon_L(\langle \Sigma', t' \parallel P'\rangle)$ if $\langle \Sigma, t \parallel P\rangle \hookrightarrow \langle \Sigma', t' \parallel P'\rangle$. The relation $\hookrightarrow_L$ guarantees that confidential data, i.e., data not below or equal-to level L, is erased as soon as it is created. We write $\hookrightarrow_L^*$ for the reflexive and transitive closure of $\hookrightarrow_L$.

In order to prove non-interference, we will establish a simulation relation between $\hookrightarrow^*$ and $\hookrightarrow^*_L$ through the erasure function: erasing all secret data and then taking evaluation steps in $\hookrightarrow_L$ is equivalent to taking steps in $\hookrightarrow$ first, and then erasing all secret values in the resulting configuration. In the rest of this section, we consider well-typed terms to avoid stuck configurations.

Proposition 1 (Many-step simulation). *If* $\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle \hookrightarrow^* \langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle$, *then it holds that* $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle) \hookrightarrow^*_L \varepsilon_L(\langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle)$.

The L-equivalence relation $\approx_L$ is an equivalence relation between configurations and their parts, defined as the equivalence kernel of the erasure function ε_L: $\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle \approx_L \langle \Sigma', \mathbf{sch}\ r_s \parallel Q\rangle$ iff $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle) = \varepsilon_L(\langle \Sigma', \mathbf{sch}\ r_s \parallel Q\rangle)$. If two configurations are L-equivalent, they agree on all data below or at level L, i.e., an attacker at level L is not able to distinguish them.

The next theorem shows the non-interference property. The configuration $\langle \Sigma, \mathbf{sch}\ []\rangle$ represents a final configuration, where the thread pool is empty and there are no more threads to run.

Theorem 1 (Termination-insensitive non-interference). *Given a computation* e, *inputs* e_1 *and* e_2, *an attacker at level* L, *runtime environments* Σ_1 *and* Σ_2, *then for all inputs* e_1, e_2 *such that* $e_1 \approx_L e_2$, *if* $\langle \Sigma_1, \mathbf{sch}\ [e\ e_1]\rangle \hookrightarrow^* \langle \Sigma'_1, \mathbf{sch}\ []\rangle$ *and* $\langle \Sigma_2, \mathbf{sch}\ [e\ e_2]\rangle \hookrightarrow^* \langle \Sigma'_2, \mathbf{sch}\ []\rangle$, *then* $\langle \Sigma'_1, \mathbf{sch}\ []\rangle \approx_L \langle \Sigma'_2, \mathbf{sch}\ []\rangle$.

This theorem essentially states that if we take two executions from configurations $\langle \Sigma_1, \mathbf{sch}\ [e\ e_1]\rangle$ and $\langle \Sigma_2, \mathbf{sch}\ [e\ e_2]\rangle$, which are indistinguishable to an attacker at level L ($e_1 \approx_L e_2$), then the final configurations for the executions $\langle \Sigma'_1, \mathbf{sch}\ []\rangle$ and $\langle \Sigma'_2, \mathbf{sch}\ []\rangle$ are also indistinguishable to the attacker ($\langle \Sigma'_1, \mathbf{sch}\ []\rangle \approx_L \langle \Sigma'_2, \mathbf{sch}\ []\rangle$). This result generalizes when constructors $Done$, $Atom$, and $Fork$ involve exception handling (see Figure 5). The reason for this lies in the fact that $catch$ and $throw$ defer all exception handling to $LIO.throw$ and $LIO.catch$, which have been proved secure in [36].

7 Case study: Classifying location data

We evaluated the trade-offs between performance, expressiveness and security through an LIO case study. We implemented an untrusted application that performs K-means clustering on sensitive user location data, in order to classify GPS-enabled cell phone into locations on a map, e.g., home, work, gym, etc. Importantly, this app is untrusted yet computes clusters for users without leaking their location (e.g., the fact that

Alice frequents the local chapter of the Rebel Alliance). K-means is a particularly interesting application for evaluating our scheduler as the classification phase is highly parallelizable—each data point can be evaluated independently.

We implemented and benchmarked three versions of this app: (i) A baseline implementation that does not use our scheduler and parallelizes the computation using Haskell's Par Monad [21]. Since in this implementation, the scheduler is not modeled using resumptions, it leverages the parallelism features of Par. (ii) An implementation in the resumption based scheduler, but pinned to a single core (therefore not taking advantage of parallelizing pure computations). (iii) A parallel implementation using the resumption-based scheduler. This implementation expresses the exact same computation as the first one, but is not vulnerable to cache-based leaks, even in the face of parallel execution on multiple cores.

We ran each implementation against one month of randomly generated data, where data points are collected each minute (so, 43200 data points in total). All experiments were run ten times on a machine with two 4-core (with hyperthreading) 2.4Ghz Intel Xeon processors and 48GB of RAM. The secure, but non-parallel implementation using resumptions performed extremely poorly. With mean 204.55 seconds (standard deviation 7.19 seconds), it performed over eight times slower than the baseline at 17.17 seconds (standard deviation 1.16 seconds). This was expected since K-means is highly parallelizable. Conversely, the parallel implementation in the resumption based scheduler performed more comparably to the baseline, at 17.83 seconds (standard deviation 1.15 seconds).

To state any conclusive facts on the overhead introduce by our library, it is necessary to perform a more exhaustive analysis involving more than a single case study.

8 Related work

Cryptosystems Attacks exploiting the CPU cache have been considered by the cryptographic community [16]. Our attacker model is weaker than the one typically considered in cryptosystems, i.e., attackers with access to a stopwatch. As a countermeasure, several authors propose partitioning the cache (e.g., [25]), which often requires special hardware. Other countermeasures (e.g. [23]) are mainly implementation-specific and, while applicable to cryptographic primitives, they do not easily generalize to arbitrary code (as required in our scenario).

Resumptions While CPS can be used to model concurrency in a functional setting [7], resumptions are often simpler to reason about when

considering security guarantees [10, 11]. The closest related work is that of Harrison and Hook [11]; inspired by a secure multi-level operating system, the authors utilize resumptions to model interleaving and layered state monads to represent threads. Every layer corresponds to an individual thread, thereby providing a notion of local state. Since we do not require such generality, we simply adapt the scheduler to context-switch the local state underlying the *LIO* monad. We believe that authors overlooked the power of resumptions to deal with timing perturbations produced by the underlying hardware. In [10], Harrison hints that resumptions could handle exceptions; in this work, we consummate his claim by describing precicely how to implement *throw* and *catch*.

Language-based IFC There is been considerable amount of literature on applying programming languages techniques to address the internal timing covert channel (e.g. [28, 33, 35, 39, 41]). Many of these works assume that the execution of a single step, i.e., a reduction step in some transition system, is performed in a single unit of time. This assumption is often made so that security guarantees can be easily shown using programming language semantics. Unfortunately, the presence of the CPU cache (or other hardware shared state) breaks this correspondence, making cache attacks viable. Our resumption approach establishes a correspondence between atomic steps at the implementation-level and reduction step in a transition system. Previous approaches can leverage this technique when implementing systems, as to avoid the reappearance of the internal timing channel.

Agat [2] presents a code transformation for sequential programs such that both code paths of a branch have the same memory access pattern. This transformation has been adapted in different works (e.g., [32]). Agat's approach, however, focuses on avoiding attacks relying on the data cache, while leaving the instruction cache unattended.

Russo and Sabelfeld [29] consider non-interference for concurrent while-like-programs under cooperative and deterministic scheduling. Similar to our work, this approach eliminates cache-attacks by restricting the use of yields. Differently, our library targets a richer programming languages, i.e., it supports parallelism, exceptions, and dynamically adjusting the granularity of atomic actions.

Secure multi-execution [8] preserves confidentiality of data by executing the same sequential program several times, one for each security level. In this scenario, cache-based attacks can only be removed in specific configurations [14] (e.g., when there are as many CPU cores as security levels).

Hedin and Sands [12] present a type-system for preventing external timing attacks for bytecode. Their semantics is augmented to incorporate history, which enables the modeling of cache effects. Zhang et al. [42] provide a method for mitigating external events when their

timing behavior could be affected by the underlying hardware. Their semantics focusses on sequential programs, wherein attacks due to the cache arise in the form of externally visible events. Their solution is directly applicable to our system when considering external events.

System security In order to achieve strong isolation, Barthe et al. [3] present a model of virtualization which flushes the cache upon switching between guest operating systems. Flushing the cache in such scenarios is common and does not impact the already-costly context-switch. Although this technique addresses attacks that leverage the CPU cache, it does not address the case where a shared resource cannot be controlled (e.g., CPU bus).

Allowing some information leakage, Kopft et al. [17] combines abstract interpretation and quantitative information-flow to analyze leakage bounds for cache attacks. Kim et al. [15] propose StealthMem, a system level protection against cache attacks. StealthMem allows programs to allocate memory that does not get evicted from the cache. StealthMem is capable of enforcing confidentiality for a stronger attacker model than ours, i.e., they consider programs with access to a stopwatch and running on multiple cores. However, we suspect that StealthMem is not adequate for scenarios with arbitrarily complex security lattices, wherein not flushing the cache would be overly restricting.

9 Conclusion

We present a library for LIO that leverages resumptions to expose concurrency. Our resumption-based approach and "instruction"- or atom-based scheduling removes internal timing leaks induced by timing perturbations of the underlying hardware. We extend the notion of resumptions to support state and exceptions and provide a scheduler which context-switches programs with such features. Though our approach eliminates internal-timing attacks that leverage hardware caches, library-level threading imposes considerable performance penalties. Addressing this, we provide programmers with a safe mean for controlling the context-switching frequency, i.e., allowing for the adjustment of the "size" of atomic actions. Moreover, we provide a primitive for spawning computations in parallel, a novel feature not previously available in IFC tools. We prove soundness of our approach and implement a simple case study to demonstrate its use. Our techniques can be adapted to other Haskell-like IFC systems beyond LIO. The library, case study, and details of the proofs can be found at [6].

Acknowledgments We would like to thank Josef Svenningsson and our colleagues in the ProSec and Functional Programming group at Chalmers for useful comments. This work was supported by the Swedish research agency VR,

STINT, the Barbro Osher foundation, DARPA CRASH under contract #N66001-10-2-4088, and multiple gifts from Google. Deian Stefan is supported by the DoD through the NDSEG Fellowship Program.

Bibliography

[1] O. Aciiçmez. Yet another microarchitectural attack:: exploiting I-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, CSAW '07. ACM, 2007.

[2] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Prog. Languages*, pages 40–53, Jan. 2000.

[3] G. Barthe, G. Betarte, J. Campo, and C. Luna. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society, june 2012.

[4] Boudol and Castellani. Noninterference for concurrent programs. In *Proc. ICALP'01*, volume 2076 of *LNCS*. Springer-Verlag, July 2001.

[5] G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1), June 2002.

[6] P. Buiras, A. Levy, D. Stefan, A. Russo, and D. Mazières. A library for removing cache-based attacks in concurrent information flow systems: Extended version. `http://www.cse.chalmers.se/~buiras/resLIO.html`, 2013.

[7] K. Claessen. A poor man's concurrency monad. *J. Funct. Program.*, May 1999.

[8] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10. IEEE Computer Society, 2010.

[9] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. of the 10th Symposium on Operating Systems Design and Implementation*, October 2012.

[10] B. Harrison. Cheap (but functional) threads. *J. of Functional Programming*, 2004.

[11] W. L. Harrison and J. Hook. Achieving information flow security through precise control of effects. In *Proc. IEEE Computer Sec. Foundations Workshop*. IEEE Computer Society, 2005.

[12] D. Hedin and D. Sands. Timing aware information flow security for a JavaCard-like bytecode. *Elec. Notes Theor. Comput. Sci.*, 141, 2005.

[13] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1996.

[14] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. of IEEE Symposium on Sec. and Privacy*. IEEE, 2011.

[15] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *Proc. of the USENIX Conference on Security Symposium*, Security'12. USENIX Association, 2012.

[16] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. of the 16th CRYPTO*. Springer-Verlag, 1996.

[17] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12. Springer-Verlag, 2012.

[18] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A World Wide Web Without Walls. In *6th ACM Workshop on Hot Topics in Networking (Hotnets)*, Atlanta, GA, November 2007.

[19] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[20] P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.

[21] S. Marlow, R. Newton, and S. L. P. Jones. A monad for deterministic parallelism. In *Proc. ACM SIGPLAN Symposium on Haskell*, 2011.

[22] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[23] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology*, CT-RSA'06. Springer-Verlag, 2006.

[24] B. Pablo and A. Russo. Lazy programs leak secrets. In *the Pre-proceedings of the 18th Nordic Conference on Secure IT Systems (NordSec)*, October 2013.

[25] D. Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptology ePrint Archive*, 2005, 2005.

[26] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[27] F. Pottier. A simple view of type-secure information flow in the π-calculus. In *In Proc. of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[28] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Sec. Foundations Workshop*, July 2006.

[29] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics (PSI)*, LNCS. Springer-Verlag, June 2006.

[30] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Proc. of Asian Computing Science Conference*, LNCS. Springer-Verlag, Dec. 2006.

[31] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN Symposium on Haskell*, pages 13–24. ACM Press, Sept. 2008.

[32] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Sec. Foundations Workshop*, July 2000.

[33] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Prog. Languages*, Jan. 1998.

[34] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

[35] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *The 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 201–213. ACM, September 2012.

[36] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457*, 2012.

[37] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *Proc. European Symp. on Research in Computer Security*, 2013.

[38] W. Swierstra. *A Functional Specification of Effects*. PhD thesis, University of Nottingham, November 2008.

[39] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3), Nov. 1999.

[40] W. H. Wong. Timing attacks on RSA: revealing your secrets through the fourth dimension. *Crossroads*, 11, May 2005.

[41] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Sec. Foundations Workshop*, June 2003.

[42] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proc. of PLDI*. ACM, 2012.

CHAPTER

FIVE

LAZY PROGRAMS LEAK SECRETS

Pablo Buiras, Alejandro Russo

Abstract. To preserve confidentiality, information-flow control restricts how untrusted code handles secret data. While promising, IFC systems are not perfect; they can still leak sensitive information via covert channels. In this work, we describe a novel exploit of *lazy evaluation* to reveal secrets in IFC systems. Specifically, we show that lazy evaluation might transport information through the *internal timing covert channel*, a channel present in systems with concurrency and shared resources. We illustrate our claim with an attack for LIO, a concurrent IFC system for Haskell. We propose a countermeasure based on restricting the implicit sharing caused by lazy evaluation.

1 Introduction

Information-flow control (IFC) permits untrusted code to safely operate on secret data. By tracking how data is disseminated inside programs, IFC can avoid leaking secrets into public channels—a policy known as non-interference [4]. Despite being promising, IFC systems are not flawless; the presence of covert channels allows attackers to still leak sensitive information.

Covert channels arise when programming language features are misused to leak information [6]. The tolerance to such channels is determined by their bandwidth and how easy it is to exploit them. For instance, the termination covert channel, which exploits divergence of programs, has a different bandwidth in systems with intermediate outputs than in batch processes [1].

Lazy evaluation is the default evaluation strategy of the purely functional programming language Haskell. This evaluation strategy has two distinctive features which can be used together to reveal secrets. Firstly, since it is a form of *non-strict evaluation*, it delays the evaluation of function/constructor arguments and let-bound identifiers until their denoted values are needed. Secondly, when the evaluation of such expressions is required, their resulting value is stored (cached) for subsequent uses of the same expression, a feature known as *sharing* or *memoisation*. This is known as *call-by-need* semantics or simply lazy evaluation. In Haskell, a *thunk*, also known as a delayed computation, is a parameterless closure created to prevent the evaluation of an expression until it is required at a later time. The process of evaluating a thunk is known as *forcing*. While lazy evaluation does not affect the denotation of expressions with respect to non-strict semantics, it affects the timing behaviour of programs. For instance, if a function argument is used more than once in the body of a function, it is almost always faster to use lazy evaluation as opposed to call-by-name, since it avoids re-evaluating every occurrence of the argument.

From a security point of view, it is unclear what type of semantics (non-strict versus strict) is desirable in order to deal with covert channels. In sequential settings, Sabelfeld and Sands [10] suggest that a non-strict semantics might be intrinsically safer than a strict one. This observation is based on the ability to exploit the *termination covert channel*. Although it could avoid termination leaks, lazy evaluation can compromise security in other ways. For instance, Rafsson et al. [9] describe how to exploit the Java (lazy) class initialisation process to reveal secrets. Not surprinsingly, lazy evaluation might also reveal secrets through the *external timing covert channel*. This channel involves externally measuring the time used to complete operations that may depend on secret data.

More interestingly, and totally unexplored until this work, lazy evaluation might transport information through the *internal timing covert*

channel. This covert channel arises by the mere presence of concurrency and shared resources. Malicious code can exploit it by setting up threads to race for a public shared resource and, depending on the secret, affecting their timing behaviour to determine the winner. With lazy evaluation in place, thunks become shared resources and forcing their evaluation corresponds to affecting the threads' timing behaviour—subsequent evaluations of previously forced thunks take practically no time.

We present an attack for *LIO* [12], a concurrent IFC system for Haskell, that leverages lazy evaluation to leak secrets. *LIO* presents countermeasures for internal timing leaks based on programming language level abstractions. Since *LIO* is embedded in Haskell as a library, lazy evaluation, as a feature that primarily affects pure values, is handled by the host language. Lazy evaluation is essentially built into Haskell's internals, hence there are no programming language-level mechanisms for inspecting or creating thunks that could be used to implement a countermeasure. Thunks for pure values are transparently injected into *LIO* computations, so the library could not be capable of explicitly considering whether they have been memoised at any given time.

This paper is organised as follows. Section 2 briefly recaps the basics of *LIO*. Section 3 presents the attack. Section 4 describes a possible countermeasure. Conclusions are drawn in Section 5.

2 *LIO*: a concurrent IFC system for Haskell

In purely functional languages, computations with side-effects are encoded as values of abstract data types called monads [8]. In Haskell, there are monads for performing inputs and outputs (monad *IO*), handling errors (monad *Error*), etc. The IFC system *LIO* is simply another monad in which security checks are performed before side-effects are performed.

The *LIO* monad keeps track of a *current label*. This label is an upper bound on the labels of all data in lexical scope. When a computation C, with current label L_C, observes an object labelled L_O, C's label is raised to the least upper bound or *join* of the two labels, written $L_C \sqcup L_O$. Importantly, the current label governs where the current computation can write, what labels may be used when creating new channels or threads, etc. For example, after reading an object O, the computation should not be able to write to a channel K if L_O is more confidential than L_K—this would potentially leak sensitive information (about O) into a less sensitive channel.

Since the current label protects all the variables in scope, in practical programs we need a way of manipulating differently-labelled data without monotonically increasing the current label. For this purpose, *LIO* provides explicit references to labelled, immutable data through a para-

```
attack :: LMVar LH Int → Labeled LH Int → LIO LH Int
attack lmv secret
  = do let thunk = [1 .. constant] :: [Int]
         -- Thread C
       forkLIO (do s ← unlabel secret
                   when (s ≢ 0) (do n ← traverse thunk
                                    when (n > 0) (return ())))
       threadDelay delay_C
         -- Thread A
       forkLIO (do n ← traverse thunk
                   when (n > 0) (putLMVar lmv 1))
         -- Thread B
       forkLIO (do threadDelay delay_B
                   putLMVar lmv 0)
       w ← takeLMVar lmv
       _ ← takeLMVar lmv
       return w
```

Fig. 1: Attack exploiting lazy evaluation

metric data type called *Labeled*. A locally accessible symbol can bind, for example, a value of type *Labeled l Int* (for some label type *l*), which contains an *Int* protected by a label different from the current one. Function *unlabel* :: *Labeled l a* → *a*[1] brings the labelled value into the current lexical scope and updates the current label accordingly.

LIO also includes IFC-aware versions of well-established synchronisation primitives known as *MVar*s [5]. A value of type *LMVar* is a mutable location that is either empty or contains a value. Function *putLMVar* fills the *LMVar* with a value if it is empty and blocks otherwise. Dually, *readLMVar* empties an *LMVar* if it is full and blocks otherwise.

3 A lazy attack for *LIO*

Figure 1 shows the attack for *LIO*. The code essentially implements an internal timing attack [11] which leverages lazy evaluation to affect the timing behaviour of threads. We assume the classic two-point lattice (of type *LH*) where security levels *L* and *H* denote public and secret data, respectively, and the only disallowed flow is the one from *H* to *L*. Function *attack* takes a public, shared *LMVar lmv*, and a labelled boolean *secret* (encoded as an integer for simplicity). The goal of *attack*

[1] Symbol :: introduces type declarations and → denotes function types.

is to return a public integer equal to $secret$, thus exposing an LIO vulnerability. In isolation, all the threads are secure. When executed concurrently, however, $secret$ gets leaked into lmv. For simplicity, we use $threadDelay\ n$, which causes a thread to sleep for n micro seconds, to exploit the race to lmv—if such an operation was not allowed, using a loop would work equally well.

The attack proceeds as follows. Threads A and B do not start running until thread C finishes. This effect can be easily achieved by adjusting the parameter $delay\_C$. The role of thread C is to force the evaluation of the list $thunk$ when the value of $secret$ is not zero ($s \not\equiv 0$). To that end, function $traverse$ goes over $thunk$, returning one of its elements. Condition $n > 0$ always holds and it is only used to force Haskell to fully evaluate the closure returned by $traverse$. Threads A and B will eventually start racing. Thread A executes the command $traverse\ thunk$ before writing the constant 1 into lmv ($putLMVar\ lmv\ 1$). Thread B delays writing 0 into lmv ($putLMVar\ \ lmv\ \ 0$) by some (carefully chosen) time $delay\_B$. If $s \not\equiv 0$, $thunk$ will have already been evaluated when thread A traverses its elements, thus taking less time than thread B's delay. As a result, value 1 is first written into lmv. Otherwise, thread B's delay is shorter than the time taken by thread A to force the evaluation of $thunk$. In this case, value 0 is first written into lmv. Variable w observes the first written value in lmv, which will coincide with the value of the secret. The precise values of parameters $constant$, $delay\_C$, and $delay\_B$ are machine-specific and experimentally determined.

The following code shows the magnification of the attack for a list of secret integers.

$$
\begin{aligned}
&magnify :: [Labeled\ LH\ Int] \rightarrow LIO\ LH\ [Int] \\
&magnify\ ss = \mathbf{do}\ lmv \leftarrow newEmptyLMVar\ L \\
&\qquad\qquad\qquad mapM\ (attack\ lmv)\ ss
\end{aligned}
$$

Function $magnify$ takes a list of secret values ss (of type $[Labeled\ LH\ Int]$). The magnification proceeds by creating the public $LMVar$ ($newEmptyLMVar\ L$) needed by the attack. Function $mapM$ sequentially applies function $attack\ lmv$ (i.e. the attack) to every element in ss and collects the results in a public list ($[Int]$).

Below, we present the final component required for the attack:

$$
\begin{aligned}
&traverse :: [a] \rightarrow LIO\ LH\ a \\
&traverse\ xs = return\ (last\ xs)
\end{aligned}
$$

This function simply returns the last element of the list given as argument.

The code for the attack can be downloaded from `http://www.cse.chalmers.se/~buiras/LazyAttack.tar.gz`.

4 Restricting sharing

We propose a countermeasure based on restricting the sharing feature of lazy evaluation. Specifically, we propose duplicating shared thunks when spawning new threads. In that manner, sharing gets restricted to the lexical scope of each thread. Thunks being forced in one thread will then not affect the timing behaviour of the others. To illustrate this point, consider the shared $thunk$ from Figure 1. If this countermeasure was implemented, forcing the evaluation of $thunk$ by thread C would not affect the time taken by thread A to evaluate $traverse\ thunk$, making the attack no longer possible. An important drawback of this approach is that there would be a performance penalty incurred by disabling sharing among threads. Benchmarking and evaluation would be necessary to determine the full extent of the overhead inherent in the technique. Presumably, programmers could restructure their programs to minimise the effect of this penalty.

As an optimisation, it is possible to only duplicate thunks denoting pure expressions. Thunks denoting side-effecting expressions can be shared across threads without jeopardising security. The reason for that relies on LIO's ability to monitor side-effects. If a thread that depends on the secret forces the evaluation of side-effecting computations, the resulting side-effects are required to agree with the IFC policy. For instance, threads with secrets in lexical scope can only force thunks that perform no public side-effects; otherwise LIO will abort the execution in order to preserve confidentiality.

To implement our approach, we propose using **deepDup**, an operation introduced by Joachim Breitner [2] to prevent sharing in Haskell. Essentially, **deepDup** takes a variable as its argument and creates a private copy of the whole heap reachable from it, effectively duplicating the argument thunk and disabling sharing between it and the original thunk. In his paper, Breitner shows how to extend Launchbury's natural semantics for lazy evaluation [7] with **deepDup**. The natural semantics is given by a relation $\Gamma : t \Downarrow \Delta : v$, which represents the fact that from the heap Γ we can reduce term t to the value v, producing a new heap Δ. It is the relation between Γ and Δ which captures heap modifications caused by memoisation. In this setting, the rule for **deepDup** is

$$\frac{\Gamma, x \mapsto e, x' \mapsto \hat{e}[y_1'/y_1 \ldots, y_n'/y_n], (y_i' \mapsto \mathbf{deepDup}\ y_i)_{i \in 1 \ldots n} : x' \Downarrow \Delta : z \qquad \mathrm{ufv}(e) = \{y_1, \ldots, y_n\} \qquad x', y_1', \ldots, y_n' \text{ fresh}}{\Gamma, x \mapsto e : \mathbf{deepDup}\ x \Downarrow \Delta : z}$$

where $\mathrm{ufv}(e)$ is the set of unguarded[2] free variables of e and $\hat{e}$ is e with all bound variables renamed to fresh variables in order to avoid vari-

[2] Function $\mathrm{ufv}(e)$ is defined as the set of free variables that are not already marked for duplication, i.e. $\mathrm{ufv}(\mathbf{deepDup}\ x) = \emptyset$, and in the rest of the cases it is inductively defined as usual.

able capture when applying substitutions. Note that **deepDup** x duplicates all the thunks reachable from x in a lazy manner: the free variables $y_1, \ldots, y_n$ are replaced with calls to **deepDup** for each variable, so these duplications will not be performed until those variables are actually evaluated. Laziness is necessary to properly handle cyclic data structures, since the duplication process would loop indefinitely if it were to eagerly copy all thunks for such structures. As explained below, this design decision has important consequences for security.

In practice, we would use this primitive every time we fork a new thread: we take the body of the new thread m_1 and the body of the parent thread m_2, and replace them with **deepDup** m_1 and **deepDup** m_2. Due to the lazy nature of the duplication performed by **deepDup**, it is necessary to duplicate both thunks, i.e., m_1 and m_2. Consider two threads A and B with current labels L and H, respectively, and suppose that they both have a pointer to a certain thunk x in the same scope. If we only duplicated the thunk in A (the public thread), thread B could evaluate parts of x depending on the secret, before they have been duplicated in thread A—recall that **deepDup** is lazy. This would cause the evaluation of the same parts of the duplicated version of x in A to go faster, thus conveying some information about the secret to thread A. In addition, note that it is not possible to determine in advance—at the time $forkLIO$ is called—which thread will raise its current label to H. Therefore, we must take care to duplicate all further references to shared thunks every time a fork occurs.

As a possible optimisation, we advise designing a data dependency analysis capable of over-approximating which expressions are shared among threads. Once the list of expressions (and their scope) has been calculated, we would proceed to instrument the code, introducing instructions that duplicate only the truly shared thunks at runtime, as opposed to duplicating every pure thunk in the body of each thread. We believe that HERMIT [3] is an appropriate tool to deploy such instrumentation as a code-to-code transformation.

5 Conclusions

We describe and implement a new way of leveraging lazy evaluation to leak secrets in LIO, a concurrent IFC system in Haskell. Beyond LIO, the attack points out a subtlety of IFC for programming languages with lazy semantics and concurrency. We propose a countermeasure based on duplicating thunks at the time of forking in order to restrict sharing among threads. For that, we propose to use the experimental Haskell package $ghc\text{-}dup$. This package provides operations that copy thunks in a lazy manner. Although convenient for preserving program semantics, such design decision has implications for security. To deal with that, our

solution requires duplicating thunks for both the newly spawned thread and its parent. As future work, we will implement the proposed countermeasure, prove soundness (non-interference), evaluate its applicability through different case studies, and introduce some optimisations to reduce the amount of duplicated thunks.

Acknowledgments We would like to thank Andrei Sabelfeld, David Sands, and the anonymous reviewers for useful comments. Special thanks to Edward Z. Yang, who mentioned the work by Joachim Breitner to us. This work was funded by the Swedish research agency VR, STINT, and the Barbro Osher foundation.

Bibliography

[1] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. of the European Symp. on Research in Computer Security (ESORICS)*. Springer-Verlag, 2008.

[2] J. Breitner. dup – Explicit un-sharing in Haskell. *CoRR*, abs/1207.2017, 2012.

[3] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the machine: a plugin for the interactive transformation of GHC core language programs. In *Proc. ACM SIGPLAN Symposium on Haskell*, 2012.

[4] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

[5] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. ACM Symp. on Principles of Prog. Languages*. ACM, 1996.

[6] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[7] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. ACM Symp. on Principles of Prog. Languages*. ACM, 1993.

[8] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[9] W. Rafnsson, K. Nakata, and A. Sabelfeld. Securing class initialization in Java-like languages. *IEEE Transactions on Dependable and Secure Computing*, 10(1), Jan. 2013.

[10] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order Symbol. Comput.*, 14(1), Mar. 2001.

[11] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Prog. Languages*, Jan. 1998.

[12] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2012.